USENIX Association

Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation

February 25–27, 2020
Santa Clara, CA, USA

ISBN 978-1-939133-13-7

Conference Organizers

External Reviewers

Message from the NSDI ’20 Program Co-Chairs

Welcome to NSDI ’20!

NSDI is traditionally the top venue for papers on networked and distributed systems, and this year we continue that tradition with an excellent program featuring yet another record number of amazing papers. The research and experiences described in this year’s program include a wide range of topics including analytics, data center network architecture, distributed systems, host networking, machine learning, modern network hardware, monitoring and diagnosis, operating systems, privacy, security, wireless networking and deployed industrial experience.

This year, we continued the changes put into place during last year’s conference, including two deadlines and the possibility of a “one shot” revision process. We received 354 submissions (79 in the Spring cycle and 275 in the Fall cycle), and we accepted a record 65 papers, resulting in an 18.3% acceptance rate. These papers were carefully selected by a panel of 57 experts spanning academia and industry.

The review process was double-blind. We had two rounds of reviews per cycle, providing papers that advanced to the second round at least five reviews. We strove to include valuable feedback in all these reviews, so we hope it benefited all authors who submitted their work. After writing reviews, we held online discussions to select papers to be discussed further at PC meetings or to be accepted without further need for discussion. We discussed 24 papers during the Spring PC meeting (conducted online via video conferencing software) and 90 papers during the 1.5-day Fall PC meeting held on the UC San Diego campus in La Jolla, Calif.

We’d like to thank the many, many people whose work was necessary to arrange this conference. Foremost, we thank all the authors who chose to send their strong work to NSDI. We also thank the program committee whose diligence, professionalism, expertise, excitement, and courtesy made the review process go smoothly and successfully. Special thanks to those of them who took on extra responsibilities beyond the considerable ones PC members already have. Thank you to Ivan Beschastnikh and Ravi Netravali for serving as poster co-chairs. We’d like to thank Sujata Banerjee and Rebecca Isaacs for handling chair conflict papers. We’d like to thank Ellen Zegura, Laurant Vanbever, and Aditya Akella for selecting the best papers and the community award paper. We’d like to thank the members of the Test of Time Awards committee: Aditya Akella, Katerina Argyraki, Sujata Banerjee, Paul Barham, Miguel Castro, Nick Feamster, Jon Howell, Arvind Krishnamurthy, Jay Lorch, Jeff Mogul, Timothy Roscoe, Srinivasan Seshan, Alex Snoeren, and Minlan Yu. We’d like to thank the NSDI steering committee as well as the co-chairs of NSDI 2019, Minlan Yu and Jay Lorch.

We would like to thank Jennifer Folkestad, who organized the in-person PC meeting on UC San Diego’s campus, the PC dinner, and then an entirely different second PC dinner on an hour’s notice after the first restaurant caught fire moments before we were to arrive.

We’re so very grateful to the USENIX staff, including Casey Henderson, Ginny Staubach, Jasmine Murcia, Jessica Kim, Michele Nelson, and Sarah TerHune, for the extraordinary levels of support they provided.

Ranjita Bhagwan, *Microsoft Research India*
George Porter, *University of California, San Diego*
NSDI ’20 Program Co-Chairs

NSDI '20: 17th USENIX Symposium on Networked Systems Design and Implementation

February 25–27, 2020

Boston, MA, USA

Expanding across time to deliver bandwidth efficiency and low latency

William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter
University of California San Diego

Abstract

Datacenters need networks that support both low-latency and high-bandwidth packet delivery to meet the stringent requirements of modern applications. We present Opera, a dynamic network that delivers latency-sensitive traffic quickly by relying on multi-hop forwarding in the same way as expander-graph-based approaches, but provides near-optimal bandwidth for bulk flows through direct forwarding over time-varying source-to-destination circuits. Unlike prior approaches, Opera requires no separate electrical network and no active circuit scheduling. The key to Opera's design is the rapid and deterministic reconfiguration of the network, piece-by-piece, such that at any moment in time the network implements an expander graph, yet, integrated across time, the network provides bandwidth-efficient single-hop paths between all racks. We show that Opera supports low-latency traffic with flow completion times comparable to cost-equivalent static topologies, while delivering up to 4× the bandwidth for all-to-all traffic and supporting up to 60% higher load for published datacenter workloads.

1 Introduction

Datacenter networks are tasked with providing connectivity between an ever-increasing number of end hosts whose link rates improve by orders of magnitude every few years. Preserving the "big-switch" illusion of full bisection bandwidth [2, 21] by augmenting the internal switching capacity of the network accordingly is increasingly cost prohibitive and likely soon infeasible [35]. Practitioners have long favored over-subscribed networks that provide all-to-all connectivity, but at only a fraction of host-link speeds [21, 41]. Such networks realize cost savings by dramatically reducing the amount of in-network capacity (in terms of both the number and rate of links and switches internal to the network fabric), providing full-speed connectivity between only a subset of hosts, and more limited capacity between others.

The catch, of course, is that any under-provisioned topology inherently biases the network toward certain workloads. Traditional over-subscribed Clos topologies only support rack-local traffic at full line rate; researchers have proposed alternate ways of deploying a limited amount of switching capacity—either through disparate link and switch technologies [31, 38, 40, 44], non-hierarchical topologies [27, 29, 42, 43], or both [20, 34]—that can deliver higher performance for published workloads [4, 39] at similar costs. Because workloads can be dynamic, many of these proposals implement reconfigurable networks that allocate link capacity in a time-varying fashion, either on a fixed schedule [34, 40] or in response to recent demand [20, 31, 44]. Unfortunately, practical reconfigurable technologies require non-trivial delay to retarget capacity, limiting their utility for workloads with stringent latency requirements.

Under-provisioned networks often incorporate some flavor of indirect traffic routing to address inopportune traffic demands; because application workloads do not always align well with the structure of the network, some traffic may transit longer, less-efficient paths. The benefits of indirection come at significant cost, however: traversing more than a single hop through the network imposes a "bandwidth tax." Said another way, x bytes sent over a direct link between two end points consume only x bytes of network capacity. If that same traffic is instead sent over k links, perhaps indirecting through multiple switches, it consumes $(k \cdot x)$ bytes of network capacity, where $(k-1)x$ corresponds to the bandwidth tax. Hence, the effective carrying capacity of a network, i.e., net the bandwidth tax, can be significantly less than its raw switching capacity; aggregate tax rates of 200–500% are common in existing proposals.

Reconfigurable networks seek to reduce the overall bandwidth tax rate of a given workload by provisioning direct links between end points with the highest demands, eliminating the tax on the largest, "bulk" flows whose completion time is gated by available network capacity, rather than propagation delay. The time required to identify such flows [31, 44] and reconfigure the network [20, 34], however, is generally orders-of-magnitude larger than the one-way delay of even an indirect route through the network, which is the main driver of completion times for small flows. Hence, dynamic networks face a fundamental trade-off between amortizing the overhead of reconfiguration against the inefficiency of sub-optimal configurations. The upshot is existing proposals are either unsuitable for latency sensitive traffic (which is frequently shunted to an entirely separate network in so-called hybrid architectures [31, 34, 38]), or pay substantial bandwidth tax to provide low-latency connectivity, especially when faced with dynamic or unpredictable workloads.

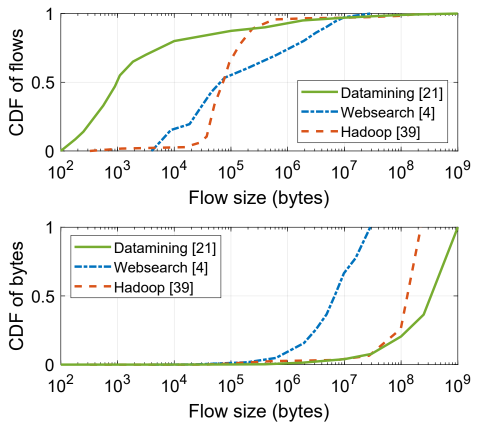


Figure 1: Published empirical flow-size distributions.

Opera is a network architecture that minimizes the bandwidth tax paid by bulk traffic—which makes up the vast majority of the bytes in today's networks [4, 39]—while ensuring low-latency delivery for the (small fraction of) traffic that cannot tolerate added delays. Opera implements a dynamic, circuit-switched topology that constantly reconfigures a small number of each top-of-rack (ToR) switch's uplinks, moving through a series of time-varying expander graphs (without requiring runtime circuit selection algorithms or network-wide traffic demand collection). Opera's ever-changing topology ensures that every pair of end points is periodically allocated a direct link, delivering bandwidth-efficient connectivity for bulk traffic, while indirecting latency-sensitive traffic over the same, low-diameter network to provide near-optimal flow completion times.

By strategically pre-configuring the assignment of rack-to-rack circuits at each instant in time such that those circuits form an expander graph, Opera can always forward low-latency traffic over an expander without waiting for any circuits to be (re-)configured. Thus, on a per-packet basis, Opera can choose to either (1) immediately send a packet over whatever static expander is currently instantiated, incurring a modest tax on this small fraction of traffic, or (2) buffer the packet and wait until a direct link is established to the ultimate destination, eliminating the bandwidth tax on the vast majority of bytes. Our simulation results show this trade-off results in up to a 4× increase in throughput for shuffle workloads compared to cost-equivalent static topologies. Moreover, for published, skewed datacenter workloads, Opera delivers an effective 8.4% bandwidth tax rate, resulting in up to a 60% increase in throughput while maintaining equivalent flow completion times across all flow sizes. We further validate the stability of this result across a range of workloads, network scales, and cost factors.

2 Network efficiency

The reality of datacenter networks is one of non-stop change: developers are continuously deploying new applications and updating existing applications, and user behavior is in a constant state of flux. As a result, operators cannot risk designing networks that support only a narrow range of workloads, and instead must choose a design that supports a wide range of workloads, applications, and user behavior.

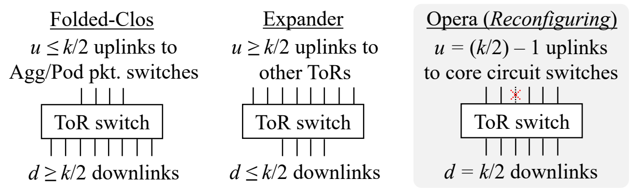


Figure 2: Oversubscribed folded-Clos networks allocate fewer uplinks than downlinks, and static expander-graph-based networks typically allocate more upward ports than downward ports. In Opera, the ToR switch is provisioned 1:1. When a circuit switch is reconfiguring, the associated ToR port cannot carry traffic through that uplink.

2.1 Workload properties

One saving grace of the need to service a wide range of workloads is the likelihood that there will, in fact, be a spectrum of needs in practice. A concrete example is the distribution of flow sizes, which is known to be highly skewed in today's networks: Figure 1 shows data published by Microsoft [4, 21] (Websearch and Datamining) and Facebook [39] (Hadoop) depicting the distributions of traffic according to individual flows (top) and total number of transmitted bytes (bottom) that we consider in this paper. The vast majority of bytes are in bulk flows, not the short, latency-sensitive ones, suggesting that to make the most out of available capacity, an ideal network must seek to minimize the bandwidth tax paid on bulk traffic while not substantially impacting the propagation delay experienced by short flows.

While there are myriad ways to measure a network's suitability for a given workload, flow completion time (FCT) is frequently offered as a useful figure of merit [14] due to its applicability across a wide range of workloads. The flow completion time of small flows is constrained by the underlying network's propagation delay. Thus, lowering the network diameter and/or reducing queuing reduces the FCT for this type of traffic. On the other hand, the FCT of bulk traffic is governed by the available capacity along a flow's path.

Because the FCT of short flows is dictated by propagation delay, such traffic is commonly referred to as "latency-sensitive" or, equivalently, "low-latency". (While applications may be equally sensitive to the FCT of larger flows, their FCT is dominated by available bandwidth.) In today's networks, flows are classified into these categories either explicitly (e.g., by application type, port number, or sender-based rules), or implicitly (e.g., by remaining flow size for shortest-remaining-time-first (SRTF) scheduling). Opera is agnostic to the manner in which traffic is classified; for our purposes latency-sensitive and short flows are synonymous. Because latency-sensitive traffic's impact on network capacity is negligible in today's workloads, it suffices to use priority queuing to ensure short flows receive unimpeded service while allowing bulk traffic to consume any remaining capacity [7, 22]. The challenge is to simultaneously provide high-capacity paths while maintaining a short path length.

2.2 The "big switch" abstraction

If cost (and practicality) were no object, a perfect network would consist of one large, non-blocking switch that connects all the end points. It is precisely such a "big switch" illusion that scale-out packet-switched network fabrics based on folded-Clos topologies [2, 21, 37] were designed to provide. These topologies rely on multiple stages of packet switches interconnected with shuffle networks. The abundance of packet switches at each stage and surfeit of links between them ensures that there is sufficient capacity to support any mixture of (admissible) inter-server communication. Proposals such as Hedera [3], pHost [19], HULL [5], NDP [24], PIAS [7], and Homa [36] introduce flow scheduling techniques that assign traffic to well-chosen paths to maximize throughput while minimizing in-network queuing when servicing a mixture of bulk and low-latency traffic.

2.3 Reduced capacity networks

While full-bandwidth "big switch" network designs are ideal in the sense that they provide operators with the maximum flexibility to deploy services, schedule jobs, and disaggregate storage and compute, they are impractical to construct at scale. Indeed, published reports confirm the largest datacenter networks in existence, while based upon folded-Clos topologies, are not fully provisioned [15, 41]. Moreover, some have observed that packet-switching technology may not be able to keep up as link rates surpass 400 Gb/s, so it is unclear how much longer the "big switch" abstraction will even be feasible [35]. Hence, researchers and practitioners alike have considered numerous ways to under-provision or "over-subscribe" network topologies.

One way to view over-subscription in a rack-based datacenter is to consider how each individual ToR switch is provisioned. Consider a scenario in which servers in a cluster or datacenter are organized into racks, each with a k-radix ToR packet switch that connects it to the rest of the network. We say that a ToR with d connected servers has d "downward" facing ports. A ToR with u ports connected to the rest of the network has u "upward" facing ports, or uplinks. (In a fully populated ToR, $d+u=k$.) In this context, we now overview existing proposals for interconnecting such racks.

Over-subscribed Fat Trees: As shown in the left-most portion of Figure 2, designers can build M:1 over-subscribed folded-Clos networks in which the network can deliver only $(1/M = u/d)$ the bandwidth of a fully-provisioned design. Common values of $(d:u)$ are between 3:1 and 5:1 [41]. The cost and bandwidth delivered in folded-Clos networks scale almost linearly according to the over-subscription factor, and so decreasing overall cost necessitates decreasing the maximum network throughput—and vice versa. Routing remains direct, however, so over-subscription does not introduce a bandwidth tax; rather, it severely reduces the available network capacity between end points in different racks. As a result, application frameworks such as MapReduce [13] and Hadoop [18] schedule jobs with locality in mind in an effort to keep traffic contained within a rack.

Expander topologies: To address the limited cross-network bandwidth available in over-subscribed Fat Trees, researchers have proposed alternative reduced-capacity network topologies based on expander graphs. In these proposals, the u uplinks from each ToR are directly connected to other ToRs, either randomly [42] or deterministically [27, 29, 43], reducing the number of switches and inter-switch links internal to the network itself. Expander-graph-based network topologies are sparse graphs with the property that there are many potential short paths from a given source to a particular destination.

Because there are no in-network switches, packets must "hop" between ToRs a number of times to reach their ultimate destination, resulting in a bandwidth tax. An expander graph with an average ToR-to-ToR hop count of L_{Avg} pays an overall bandwidth tax rate of $(L_{Avg}-1)\times$ in expectation because individual packets must indirect across a number of in-network links. The average path lengths for large networks can be in the range of 4–5 hops, resulting in a bandwidth tax rate of 300–400%. Moreover, a recent proposal [29] employs Valiant load balancing (VLB)—which imposes an additional level of explicit indirection—to address skewed traffic demands, doubling the bandwidth tax in some circumstances. One way that expanders counter-act their high bandwidth tax rate is by over-provisioning: ToRs in expander topologies typically have more upward-facing ports than down ($u > d$, as shown in the center of Figure 2)—and, hence, far more upward-facing ports than over-subscribed Fat Trees—which provides more in-network capacity. Said another way, the impact of the bandwidth tax is reduced by a factor of u/d.

Reconfigurable topologies: In an effort to reduce the bandwidth tax, other proposals rely on some form of reconfigurable link technology, including RF [28, 45], free-space optical [20, 23], and circuit switching [16, 31, 38, 40, 44]. Most reconfigurable topologies dynamically establish end-to-end paths within the network core in response to traffic demand, although RotorNet [34] employs a fixed, deterministic schedule. In either case, these networks establish and tear down physical-layer links over time. When the topology can be matched to the demand—and setting aside latency concerns—traffic can be delivered from source to destination in a single hop, avoiding any bandwidth tax. In some cases, similar to expander-based topologies, they employ 2-hop VLB [34, 40], resulting in a 100% bandwidth tax rate.

A fundamental limitation of any reconfigurable topology, however, is that during the time a link/beam/circuit (for simplicity we will use the latter term in the remainder of the paper) is being provisioned, it cannot convey data. Moreover, most proposals do not provision links between all sources and destinations at all times, meaning that traffic may incur significant delay as it waits for the appropriate circuit to be provisioned. For existing proposals, this end-to-end delay is on the order of 10–100s of milliseconds. Hence, previous proposals for reconfigurable network topologies rely on a distinct, generally packet-switched, network to service latency-sensitive traffic. The requirement for a separate network built using a different technology is a significant practical limitation and source of cost and power consumption.

3 Design

We start with an overview of our design before working through an example. We then proceed to describe how we construct the topology of a given network, how routes are chosen, how the network moves through its fixed set of configurations, and address practical considerations like cabling complexity, switching speeds, and fault tolerance.

3.1 Overview

Opera is structured as a two-tier leaf-spine topology, with packet-switched ToRs interconnected by reconfigurable circuit switches as shown in Figure 5. Each of a ToR's u uplinks are connected to one of u circuit switches, and each circuit switch has a number of ports equal to the number of ToRs in the network. Opera's design is based around two fundamental starting blocks that follow directly from the requirements for small network diameter and low bandwidth tax.

Expansion for short paths: Because the FCT of short, latency-sensitive flows is gated by end-to-end delay, we seek a topology with the lowest possible expected path length. Expander-based topologies are known to be ideal [27]. Expanders also have good fault-tolerance properties; if switches or links fail, there are likely to be alternative paths that remain. Thus, to efficiently support low-latency traffic, we require a topology with good expansion properties at all times.

Reconfigurability to avoid the bandwidth tax: A fully-connected graph (i.e. full mesh) could avoid a bandwidth tax entirely, but is infeasible to construct at scale. Rather than providing a full mesh in space, reconfigurable circuit switches offer the ability to establish, over time, direct one-hop paths between every rack pair using a relatively small number of links. Because bulk flows can generally amortize modest reconfiguration overheads if they result in increased throughput, we incorporate reconfigurability into our design to minimize the bandwidth tax on bulk traffic.

Opera combines the elements of expansion and reconfigurability to efficiently (and simultaneously) serve both low-latency and bulk traffic with low FCTs. Similar to RotorNet [34], Opera incorporates reconfigurable circuit switches that cyclically set up and tear down direct connections between ToRs, such that after a "cycle time" of connections, every ToR has been connected to every other ToR. We leverage ToR-uplink parallelism to stagger the reconfigurations of multiple circuit switches, allowing "always-on" (albeit ever-changing) multi-hop connectivity between all ToR pairs.

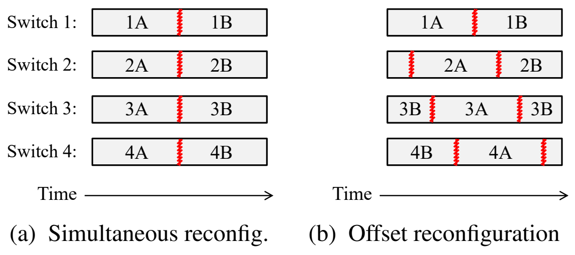


Figure 3: Reconfiguring all switches in unison (a) leads to periodic disruptions; staggered reconfigurations (b) ensure some paths are always available.

Critically, the combination of circuits at any time forms an expander graph. Thus, during a single cycle, every packet has a choice between waiting for a bandwidth-tax-avoiding direct connection, or being immediately sent over a multi-hop path through the time-varying expander. The end result is a single fabric that supports bulk and low-latency traffic as opposed to two separate networks used in hybrid approaches. As we will show, Opera does not require any runtime selection of circuits or system-wide collection of traffic demands, vastly simplifying its control plane relative to approaches that require active circuit scheduling, such as ProjecToR [20] and Mordia [38]. We leave to future work the possibility (and complexity) of adjusting Opera's matchings over long timescales to, for example, adapt to diurnal traffic patterns.

3.1.1 Eliminating reconfiguration disruptions

Circuit switches impose a technology-dependent reconfiguration delay, necessitating that flows be re-routed before reconfiguration. Even in a network with multiple circuit switches, if all switches reconfigure simultaneously (Figure 3a), the global disruption in connectivity requires routes to reconverge. For today's switching technologies, this would lead to traffic delays that could severely impact the FCTs of short, latency-sensitive flows. To avoid this scenario and allow for low-latency packet delivery, Opera offsets the reconfigurations of circuit switches. For example, in the case of small topologies with few switches, at most one switch may be reconfiguring at a time (Figure 3b), allowing flows traversing a circuit with an impending reconfiguration to be migrated to other circuits that will remain active during that time period (for large-scale networks with many circuit switches, it is advantageous to reconfigure more than one switch at a time as described in Appendix C). As a result, while Opera is in near-constant flux, changes are incremental and connectivity is continuous across time.

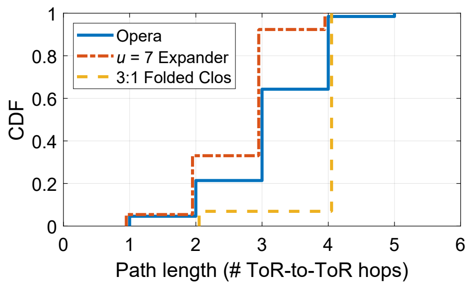


Figure 4: CDF of path lengths for equal-cost 648-host Opera, 650-host $u = 7$ expander, and 648-host 3:1 folded-Clos networks. (CDFs staggered slightly for clarity.)

3.1.2 Ensuring good expansion

While offsetting reconfigurations guarantees continuous connectivity, it does not, by itself, guarantee complete connectivity. Opera must simultaneously ensure that (1) multi-hop paths exist between all racks at every point in time to support low-latency traffic, and (2) direct paths are provisioned between every rack-pair over a fixed period of time to support bulk traffic with low bandwidth tax. We guarantee both by implementing a (time-varying) expander graph across the set of circuit switches.

In Opera, each of a ToR's u uplinks is connected to a (rotor) circuit switch [33] that, at any point in time, implements a (pre-determined) random permutation between input and output ports (i.e., a "matching"). The inter-ToR network topology is then the union of u random matchings, which, for $u \geq 3$, results in an expander graph with high probability [6]. Moreover, even if a switch is reconfiguring, there are still $u-1$ active matchings, meaning that if $u \geq 4$, the network will still be an expander with high probability, no matter which switch is reconfiguring. In Opera, we let $u = k/2$ where k is $O(10)$ to $O(100)$ ports for today's packet switches (depending on the configured per-port bandwidth).

Figure 4 shows the distribution of path lengths in one example 648-host network considered in our evaluation, where $u = 6$. Opera's path lengths are almost always substantially shorter than those in a Fat Tree that connects the same number of hosts, and only marginally longer than an expander with $u = 7$ which we argue later has similar cost, but performs poorly for certain workloads. Clearly, ensuring good expansion alone is not an issue with modest switch radices. However, Opera must also directly connect each rack pair over time. We achieve this by having each switch cycle through a set of matchings; we minimize the total number of matchings (and thus the time required to cycle through them) by constructing a disjoint set.

3.2 Example

Figure 5 depicts a small-scale Opera network. Each of the eight ToRs has four uplinks to four different circuit switches (with one potentially down due to reconfiguration at any particular moment). By forwarding traffic through those ToRs, they can reach any ToRs to which they, in turn, are connected. Each circuit switch has two matchings, labeled A and B (note that all matchings are disjoint from one another). In this example topology, any ToR-pair can communicate by utilizing any set of three matchings, meaning complete connectivity is maintained regardless of which matchings happen to be implemented by the switches at a given time. Figure 5 depicts two network-wide configurations. In Figure 5a switches 2–4 are implementing matching A, and in Figure 5b, switches 2–4 implement matching B. In both cases switch 1 is unavailable due to reconfiguration.

In this example, racks 1 and 8 are directly connected by the configuration shown in Figure 5b, and so the lowest bandwidth-tax way to send bulk data from 1 to 8 would be to wait until matching B is instantiated in switch 2, and then to send the data through that circuit; such traffic would arrive at ToR 8 in a single hop. On the other hand, low-latency traffic from ToR 1 to ToR 8 can be sent immediately, e.g. during the configuration shown in Figure 5a, and simply take a longer path to get to ToR 8. The traffic would hop from ToR 1 to ToR 6 (via switch 4), then to ToR 8 (via switch 2), and incur a 100% bandwidth tax. Although not highlighted in the figure, similar alternatives exist for all rack pairs.

3.3 Topology generation

The algorithm to generate an N-rack Opera topology is as follows. First, we randomly factor a complete graph (i.e. $N \times N$ all-ones matrix) into N disjoint (and symmetric) matchings. Because this factorization can be computationally expensive for large networks, we employ graph lifting to generate large factorizations from smaller ones. Next, we randomly assign the N matchings to u circuit switches, so that each switch has N/u matchings assigned to it. Finally, we randomly choose the order in which each switch cycles through its matchings. These choices are fixed at design time, before the network is put into operation; there is no topology computation during network operation.

Because our construction approach is random, it is possible (although unlikely) that a specific Opera topology realization will not have good expander properties at all points across time. For example, the combination of matchings in a given set of $u-1$ switches at a particular time may not constitute an expander. In this case, it would be trivial to generate and test additional realizations at design time until a solution with good properties is found. This was not necessary in our experience, as the first iteration of the algorithm always produced a topology with near-optimal properties. We discuss the properties of these graphs in detail in Appendix E.

3.4 Forwarding

We are now left to decide how to best serve a given flow or packet: (1) send it immediately over multi-hop expander paths and pay the bandwidth tax (we refer to these as "indirect" paths), or (2) delay transmission and send it over one-hop paths to avoid the bandwidth tax (we refer to these as "di-

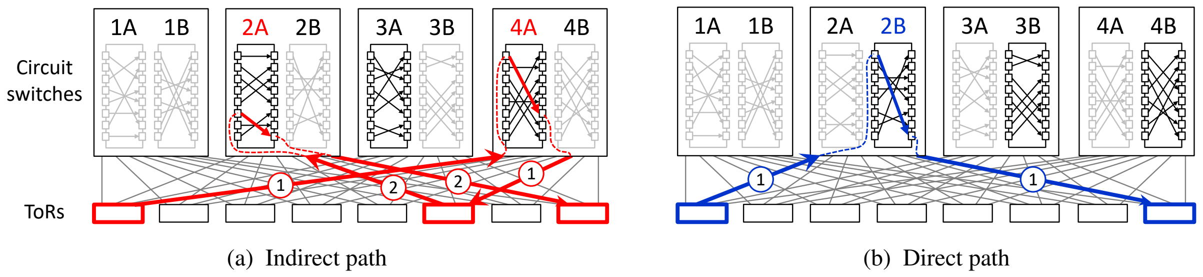


Figure 5: An Opera topology with eight ToR switches and four rotor circuit switches (from RotorNet [34]). Two different paths from rack 1 to rack 8 are highlighted: (a) a two-hop path in red, and (b) a one-hop path in blue. Each direct inter-rack connection is implemented only once per configuration, while multi-hop paths are available between each rack-pair at all times.

rect" paths). For skewed traffic patterns that, by definition, leave spare capacity in the network, two-hop paths based on Valiant load balancing can be used to carry bulk traffic in Opera. Our baseline approach is to make the decision based on flow size. Since the delay in waiting for a direct path can be an entire cycle time, we only let flows that are long enough to amortize that delay use direct paths, and place all other traffic on indirect paths. However, we can do even better if we know something about application behavior. Consider an all-to-all shuffle operation, where a large number of hosts simultaneously need to exchange a small amount of data with one another. Although each flow is small, there will be significant contention, extending the flow completion time of these flows. Minimizing bandwidth tax is critical in these situations. With application-based tagging, Opera can route such traffic over direct paths.

3.5 Synchronization

Opera employs reconfigurable circuit switches, and so its design requires a certain level of synchronization within the system to operate correctly. In particular, there are three synchronization requirements that must be met: (1) ToR switches must know when core circuit switches are reconfiguring, (2) ToR switches must update their forwarding tables in sync with the changing core circuits, and (3) end hosts must send bulk traffic to their local ToR only during the timeslots when the ToR is directly connected to the destination (to prevent excessive queueing in the ToR). In the first case, since each ToR's uplink is connected directly to one of the circuit switches, the ToR can monitor the signal strength of the transceiver attached to that link to re-synchronize with the circuit switch. Alternatively, the ToR could rely on IEEE 1588 (PTP), which can synchronize switches to within ± 1 µs [1]. For low-latency traffic, end hosts simply transmit packets immediately, without any coordination or synchronization. For bulk traffic, end hosts transmit when polled by their attached ToR. To evaluate the practicality of this synchronization approach, we built a small-scale prototype based on a programmable P4 switch, described in Section 6.

Opera can tolerate arbitrary bounds on (de-)synchronization by introducing "guard bands" around each configuration, in which no data is sent to ensure the network is configured as expected when transmissions do occur. To analyze the impact of guard bands, we hold the circuit timings constant and reduced the effective time of the slot during which data can be transmitted. Each µs of guard time contributes a 1% relative reduction in low-latency capacity and a 0.2% reduction for bulk traffic. In practice, if any component becomes de-synchronized beyond the guard-band tolerance, it can simply be declared failed (see Section 3.6.2).

3.6 Practical considerations

While Opera's design draws its power from graph-theoretic underpinnings, it is also practical to deploy. Here, we consider two real-world constraints on networks.

3.6.1 Cabling and switch complexity

Today's datacenter networks are based on folded-Clos topologies which use perfect-shuffle cabling patterns between tiers of switches. While proposals for static expander graphs alter that wiring pattern [42] leading to concerns about cabling complexity, Opera does not. In Opera, the interconnection complexity is contained within the circuit switches themselves, while the inter-switch cabling remains the familiar perfect shuffle. In principle, Opera can be implemented with a variety of electronic or optical circuit switch technologies. We focus on optical switching for our analysis due to its cost and data-rate transparency benefits. Further, because each circuit switch in Opera must only implement N/u matchings (rather than $O(N!)$), Opera can make use of optical switches with limited configurability such as those proposed in RotorNet [34], which have been demonstrated to scale better than optical crossbar switches [17, 33].

3.6.2 Fault tolerance

Opera detects, shares, and recovers from link, ToR, and circuit switch failures using common routing protocol practices. We take advantage of Opera's cyclic connectivity to detect and communicate failures: each time a new circuit is configured, the ToR CPUs on each end of the link exchange a short sequence of "hello" messages (which contain information of new failures, if applicable). If no hello messages are received within a configurable amount of time, the ToR marks the link in question as bad. Because all ToR-pair connections are established every cycle, any ToR that remains connected

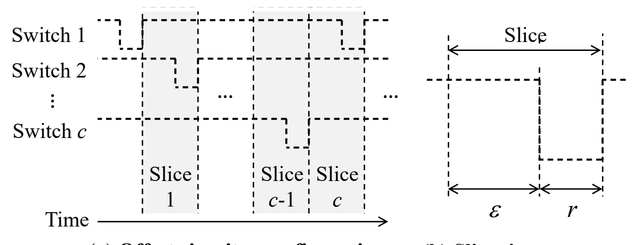


Figure 6: (a) A set of c circuit switches with offset reconfigurations forms a series of topology slices. (b) The time constants associated with a single slice: ε is the worst-case end-to-end delay for a low-latency packet to traverse the network and r is the circuit switch reconfiguration delay.

to the network will learn of any failure event within two cycles (<10ms). Upon receiving information of a new failure, a ToR recomputes and updates its routing tables to route around failed components.

4 Implementation

Here, we describe the implementation details of Opera. To ground our discussion, we refer to an example 108-rack, 648-host, $k = 12$ topology (we evaluate this topology along with a larger one in Section 5).

4.1 Defining bulk and low-latency traffic

In Opera, traffic is defined as low-latency if it cannot wait until a direct bandwidth-efficient path becomes available. Thus the division between low-latency and bulk traffic depends on the rate at which Opera's circuit switches cycle through direct matchings. The faster Opera steps through these matchings, the lower the overhead for sending traffic on direct paths, and thus the larger the fraction of traffic that can utilize these paths. Two factors impact cycle speed: circuit amortization and end-to-end delay.

Circuit amortization: The rate at which a circuit switch can change matchings is technology dependent. State-of-the-art optical switches with the large port counts needed for practical datacenter deployment have reconfiguration delays on the order of 10 µs [20, 34, 38]. A 90% amortization of this delay would limit circuit reconfigurations to every 100 µs. In Opera, each switch cycles through N/u matchings, which could range from 10 matchings for small networks (e.g. $N = 320$ racks and $u = 32$ uplinks) to 32 matchings for larger networks (e.g. $N = 4096$ racks and $u = 128$ uplinks). This means any flow than can amortize a 1–3 ms increase in its FCT could take the bandwidth-efficient direct paths (and shorter flows would take indirect paths).

End-to-end delay: Perhaps surprisingly, a second timing constraint, end-to-end delay, has a larger impact on cycle time. In particular, consider a low-latency packet that is emitted from a host NIC. At the first ToR, the packet is routed toward its destination, and in general, at each hop along the way, each ToR routes the packet along an expander-graph path. If, during the packet's journey, the circuit topology changes, it is possible the packet could be caught in a loop or redirected along a sub-optimal path. Dropping the packet immediately (and expecting the sender to resend it) would significantly delay the flow completion time of that flow.

Our approach, depicted in Figure 6, to avoid the problems described above, requires that subsequent circuit reconfigurations be spaced by at least the sum of the end-to-end delay under worst-case queuing, ε, and the reconfiguration delay, r. We refer to this time period ε+r as a "topology slice". Any packets sent during a slice are not routed through the circuit with an impending reconfiguration during that slice. This way, packets always have at least ε time to make it through the network before a switch reconfigures.

The parameter ε depends on the worst-case path length (in hops), the queue depth, the link rate, and propagation delay. Path length is a function of the expander, while the data rate and propagation delay are fixed; the key driver of ε is the queue depth. As explained in the following section, we choose a shallow queue depth of 24 KB (8 1500-byte full packets + 187 64-byte headers). When combined with a worst-case path length of 5 ToR-to-ToR hops (Figure 4), 500-ns propagation delay per hop (100 meters of fiber), and 10-Gb/s link speed, we set ε to 90 µs. In our example 108-rack network, there are 6 circuit switches, meaning the inter-reconfiguration period of a single switch is 6ε, yielding a duty cycle of 98%. Further, our example network has N/u = 108/6 = 18 matchings per switch, yielding a cycle time of $N \times \varepsilon$ = 10.8 ms. We use this cycle time of 10.8 ms in our simulations in Section 5. For these time constants, flows $\geq$15 MB will have an FCT well within a factor of 2 of their ideal (link-rate-limited) FCT. As we will show in Section 5, depending on traffic conditions, shorter flows may benefit from direct paths as well.

4.2 Transport protocols

Opera requires transport protocols that can (1) immediately send low-latency traffic into the network, while (2) delaying bulk traffic until the appropriate time. To avoid head-of-line blocking, NICs and ToRs perform priority queuing. Our design replaces the typical TCP stack with the protocols below, but keeps the familiar sockets application interface.

4.2.1 Low-latency transport

As discussed in the previous section, minimizing the cycle time is predicated on minimizing the queue depth for low-latency packets at ToRs. The recently proposed NDP protocol [24] is a promising choice because it achieves high throughput with very shallow queues. We find that 12-KB queues work well for Opera (each port has an additional equal-sized header queue). NDP also has other beneficial characteristics for Opera, such as zero-RTT convergence and no packet metadata loss to eliminate RTOs. Despite being designed for fully-provisioned folded Clos networks, we find in simulation that NDP works well with minimal modification in

Opera, despite Opera's continuously-varying topology. Other transports, like the recently proposed Homa protocol [36], may also be a good fit for low-latency traffic in Opera, but we leave this to future work.

4.2.2 Bulk transport

Opera's bulk transport protocol is relatively simple. We draw heavily from the RotorLB protocol proposed in RotorNet [34], which buffers traffic at end hosts until direct connections to the destination are available. When bulk traffic is heavily skewed, and there is necessarily spare capacity elsewhere in the network, RotorLB automatically transitions to using two-hop routing (i.e. Valiant load balancing) to improve throughput. Unlike low-latency traffic, which can be sent at any time, bulk traffic admission is coordinated with the state of the circuit switches, as described in Section 3.5. In addition to extending RotorLB to work with offset reconfigurations, we also implemented a NACK mechanism to handle cases where large bursts of priority-queued low-latency traffic can cause bulk traffic queued at the ToR to be delayed beyond the transmission window and dropped at the ToR. Retransmitting a small number of packets does not significantly affect the FCT of bulk traffic. Unlike TCP, RotorLB does not rely on retransmission timeouts, which could otherwise cause bandwidth throttling for bulk traffic.

4.3 Packet forwarding

Opera relies on ToR switches to route packets along direct or multi-hop paths depending on the requested network service model. We implement this routing functionality using the P4 programming language. Each ToR switch has a built-in register that stores the current network configuration, updated either in-band or via PTP. When a packet arrives at the first ToR switch, the packet's metadata is updated with the value of the configuration register. What happens next, and at subsequent ToR switches, depends on the value of the DSCP field. If that field indicates a low-latency packet, the switch consults a low-latency table to determine the next hop along the expander path, and then forwards the packet out that port. If the field indicates bulk traffic, the switch consults a bulk traffic table which indicates which circuit switch—if any—provides a direct connection, and the packet is forwarded to that port. We measure the amount of in-switch memory required to implement this P4 program for various datacenter sizes in Section 6.2.

5 Evaluation

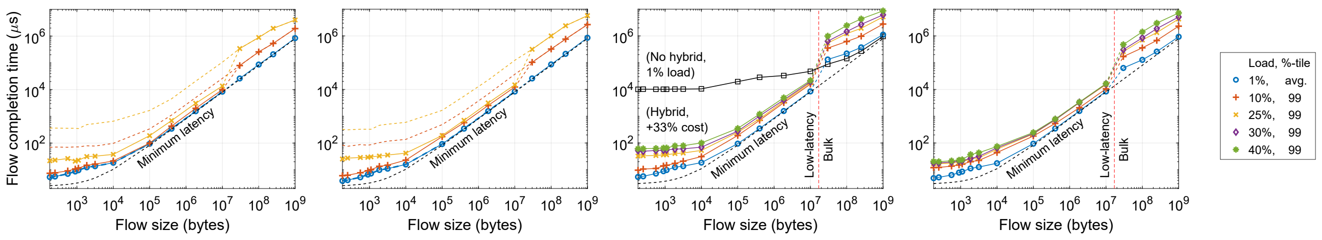
We evaluate Opera in simulation. Initially, we focus on a concrete 648-host network, comparing to cost-equivalent folded-Clos, static expander, non-hybrid RotorNet, and (non-cost-equivalent) hybrid RotorNet networks. We then validate against a range of network sizes, skewed workloads, and underlying cost assumptions. We use the `htsim` packet simulator [26], which was previously used to evaluate the NDP protocol [24], and extend it to model static expander networks and dynamic networks. We ported our RotorNet simulator [34] to `htsim`, first validating its performance against prior results. We also modify NDP to handle $<$1500 byte packets, which is necessary for some workloads considered. Both the folded-Clos and static expander use NDP as the transport protocol. Opera and RotorNet use NDP to transport low-latency traffic and RotorLB for bulk traffic. Because Opera explicitly uses priority queuing, we simulate the static networks with idealized priority queuing where appropriate to maintain a fair comparison. Following prior work [20, 29], we set the link bandwidth to 10 Gb/s. We use a 1500-byte MTU and set the propagation delay to 500 ns between ToRs (equivalent to 100 meters of fiber).

5.1 Real-world traffic

We start by considering Opera's target scenario, a workload with an inherent mix of bulk and low-latency traffic. Here we consider the Datamining workload from Microsoft [21], and use a Poisson flow-arrival process to generate flows. We vary the Poisson rate to adjust the load on the network, defining load relative to the aggregate bandwidth of all host links (i.e., 100% load means all hosts are driving their edge links at full capacity, an inadmissible load for any over-subscribed network). As shown in the top portion of Figure 1, flows in this workload range in size from 100 bytes to 1 GB. We use Opera's default configuration to decide how to route traffic: flows $<$15 MB are treated as low-latency and are routed over indirect paths, while flows $\geq$15 MB are treated as bulk and are routed over direct paths.

Figure 7 shows the performance of Opera as well as cost-comparable 3:1 folded-Clos and $u = 7$ static expander networks for various offered loads. We also compared to a hybrid RotorNet which faces one of the six ToR uplinks to a multi-stage packet switched network to accommodate low-latency traffic (for 1.33$\times$ the cost), and a cost-equivalent non-hybrid RotorNet with no packet switching above the ToR. Appendix B discusses the tradeoffs for a hybrid RotorNet in more detail. We report the 99th percentile FCT except in the case of 1% load, where the variance in the tail obscures the trend and so report the average instead. Note that Opera priority queues all low-latency flows, while by default the static networks do not. For fairness, we also present the expander and folded Clos with "ideal" priority queuing—that is, removing all flows $\geq$15 MB. For reference, we also plot the minimum achievable latency in each network, derived from the end-to-end delay and link capacity.

The static networks start to saturate past 25% load: folded Clos have limited network capacity, and expanders have high bandwidth tax. Opera, on the other hand, is able to service 40% load despite having lower innate capacity than the cost-comparable expander. Opera offloads bulk traffic onto bandwidth-efficient paths, and only pays bandwidth tax on the small fraction (4%) of low-latency traffic that transits indirect paths, yielding an effective aggregate bandwidth tax of 8.4%



(a) 3:1 folded Clos (b) $u = 7$ expander (c) RotorNet (d) Opera

Figure 7: FCTs for the Datamining workload. All networks are cost comparable except hybrid RotorNet, which is 1.33× more expensive. In (a) and (b), dashed lines are without priority queuing, and solid lines are with ideal priority queuing.

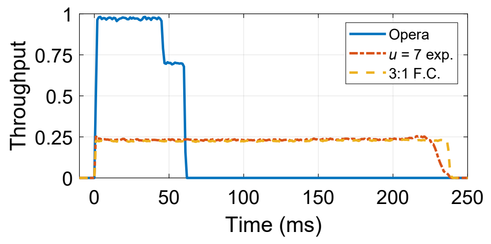


Figure 8: Network throughput over time for a 100-KB all-to-all Shuffle workload. Opera carries all traffic over direct paths, greatly increasing throughput. (The small "step" down in Opera's throughput around 50 ms is due to some flows taking one additional cycle to finish.)

for this workload. Hybrid RotorNet, even with 1/6th of its core capacity packet-switched (for 33% higher cost than the other networks), delivers longer FCTs than Opera for short flows at loads >10%. A non-hybrid (i.e. all-optical-core) RotorNet is cost-equivalent to the other networks, but its latency for short flows is three orders of magnitude higher than the other networks, as shown in Figure 7c.

5.2 Bulk traffic

Opera's superiority in the mixed case stems entirely from its ability to avoid paying bandwidth tax on the bulk traffic. We highlight this ability by focusing on a workload in which all flows are routed over direct paths. We consider an all-to-all shuffle operation (common to MapReduce style applications), and choose the flow size to be 100 KB based on the median inter-rack flow size reported in a Facebook Hadoop cluster [39] (c.f. Figure 1). Here we presume the application tags its flows as bulk, so we do not employ flow-length based classification; i.e., Opera does not indirect any flows in this scenario. We let all flows start simultaneously in Opera, as RotorLB accommodates such cases gracefully, and stagger flow arrivals over 10 ms for the static networks, which otherwise suffer from severe startup effects. Because the shuffle operation correlates the start times of all flows, this workload can drive the network to 100% utilization.

Figure 8 shows the delivered bandwidth over time for the different networks. The limited capacity of the 3:1 Clos and high bandwidth tax rates of the expander significantly extend the FCT of the shuffle operation, yielding 99th-percentile FCTs of 227 ms and 223 ms, respectively. Opera's direct paths are bandwidth-tax-free, allowing higher throughput and reducing the 99th-percentile FCT to 60 ms.

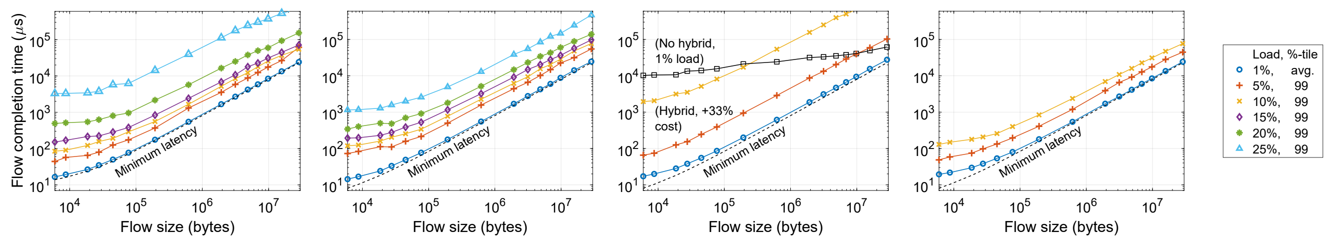
5.3 Only low-latency flows

Conversely, workloads in which all flows are routed over indirect low-latency paths represents the worst case for Opera, i.e., it always pays a bandwidth tax. Given our 15 MB threshold for bulk traffic, it is clear from the bottom portion of Figure 1 that the Websearch workload [4] represents such a case. A lower threshold would avoid the bandwidth tax, but would require a shorter cycle time to prevent a significant increase in FCT for these short "bulk" flows.

Figure 9 shows the results for the Websearch workload, again under a Poisson flow arrival process. As before, the cost-equivalent all-optical RotorNet suffers from long FCTs. Hybrid RotorNet (with 1/6th of its capacity packet switched for 33% higher cost) can only admit just over 10% load, at which point the throughput saturates. At more than 5% load, its FCTs are significantly higher than the other networks. All other networks provide equivalent FCTs across all flow sizes for loads at or below 10%, at which point Opera is not able to admit additional load. Both the 3:1 folded Clos and expander saturate (slightly) above 25% load, but at that point both deliver FCTs nearly 100× worse than at 1% load. While Opera forwards traffic analogous to the expander in this scenario, it has only 60% the capacity and pays an additional 41% bandwidth tax due to its longer expected path length.

5.4 Mixed traffic

To drive home Opera's ability to trade off low-latency capacity against lower effective bandwidth taxes, we explicitly combine the Websearch (low-latency) and Shuffle (bulk) workloads from above in varying proportions. Figure 10 shows the aggregate network throughput as a function of Websearch (low-latency) traffic load, defined as before as a fraction of the aggregate host link capacity. We see that for low Websearch load, Opera delivers up to 4× more throughput than the static topologies. Even at 10% Websearch load (near its maximum admissible load), Opera still delivers almost 2× more throughput. In essence, Opera "gives up" a factor of 2 in low-latency capacity (due to its relatively under-



(a) 3:1 folded Clos (b) $u = 7$ expander (c) RotorNet (d) Opera

Figure 9: FCTs for the Websearch workload. All networks are cost comparable except hybrid RotorNet, which is 1.33× more expensive. Opera carries all traffic over indirect paths, and supports up to 10% low-latency traffic load with near-equivalent FCTs to the 3:1 folded Clos and $u = 7$ expander.

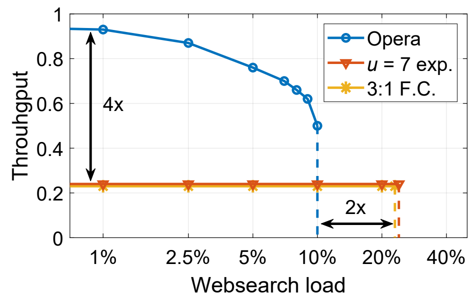


Figure 10: Network throughput vs. Websearch traffic load for a combined Websearch/Shuffle workload.

provisioned ToRs) to gain a factor of 2–4 in bulk capacity from its vastly lower effective bandwidth tax.

Facebook has reported that only 25% of traffic is attributed to bulk-dominated Hadoop jobs, but also that the total average network load is less than 1% [39]. Even if the other 75% of traffic was solely low-latency flows, Opera can accommodate this light overall load with little-to-no degradation in FCT while significantly improving throughput for Hadoop traffic, which has a high momentary peak load due to the correlated start times of constituent flows.

5.5 Fault tolerance

Next, we demonstrate Opera's ability to maintain and re-establish connectivity in the face of component failures by injecting random link, ToR, and circuit switch failures into the network. We then step through the topology slices and record (1) the number of ToR pairs that were disconnected in the worst-case topology slice and (2) the number of unique disconnected ToR pairs integrated across all slices. Figure 11 shows that Opera can withstand about 4% of links failing, 7% of ToRs failing, or 33% (2 out of 6) of circuit switches failing without suffering any loss in connectivity. Opera's robustness to failure stems from the good fault tolerance properties of expander graphs. As discussed in Appendix F, Opera has better fault tolerance than a 3:1 folded Clos, and is less tolerant than the $u = 7$ expander (which has higher fanout). Maintaining connectivity under failure does require some degree of path stretch in Opera; Appendix F discusses this in more detail.

5.6 Network scale and cost sensitivity

Finally, we examine Opera's relative performance across a range of network sizes and cost assumptions. We introduce a parameter α, which is defined following [29] to be the cost of an Opera "port" (consisting of a ToR port, optical transceiver, fiber, and circuit switch port) divided by the cost of a static network "port" (consisting of a ToR port, optical transceiver, and fiber). A full description of this cost-normalization method is presented in Appendix A. If $\alpha > 1$ (i.e. circuit switch ports are not free) then a cost-equivalent static network can use the extra capital to purchase more packet switches and increase its aggregate capacity.

We evaluated workloads previously analyzed in [29] using `htsim`: (1) hot rack, which is a highly skewed workload where one rack communicates with one other rack; (2) skew[0.1,1], (10% of racks are hot [29]), (3) skew[0.2,1] (20% hot); and (4) host permutation, where each host sends to one other non-rack-local host. For each workload, we considered a range of relative Opera port costs (reallocating any resulting cost savings in the static networks to increase their capacity). We considered both $k = 12$ and $k = 24$ ToR radices, corresponding to 648-host and 5,184-host networks. Figure 12 shows the results for $k = 24$; the $k = 12$ case has nearly identical cost-performance scaling and is presented in Appendix D, along with path length scaling analysis.

The throughput of the folded Clos topology is independent of traffic pattern, whereas the throughput of the expander topology decreases as workloads become less skewed. Opera's throughput initially decreases with a decrease in skew, then increases as the traffic becomes more uniform. As long as $\alpha < 1.8$ (Opera's circuit switch ports cost less than a packet switch port populated with an optical transceiver), Opera delivers higher throughputs than either an expander or folded Clos for permutation traffic and moderately skewed traffic (e.g. 20% of racks communicating). In the case of a single hot rack, Opera offers comparable performance to a static expander. In the case of shuffle (all-to-all) traffic, Opera delivers 2× higher throughput than either the expander or folded Clos even for $\alpha = 2$. As discussed further in Appendix A, we believe $\alpha = 1.3$ is achievable today with certain optical switching technologies.

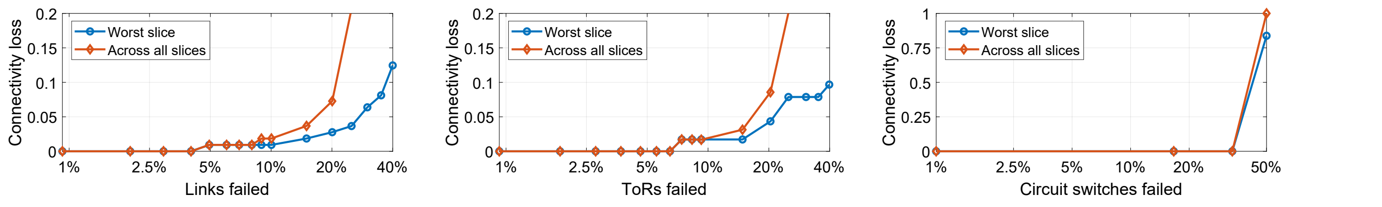


Figure 11: Fault tolerance in a 648-host, 108-rack Opera network with 6 circuit switches and $k = 12$ port ToRs. Connectivity loss is the fraction of disconnected ToR pairs. In cases involving ToR failures, connectivity loss refers to non-failed ToRs.

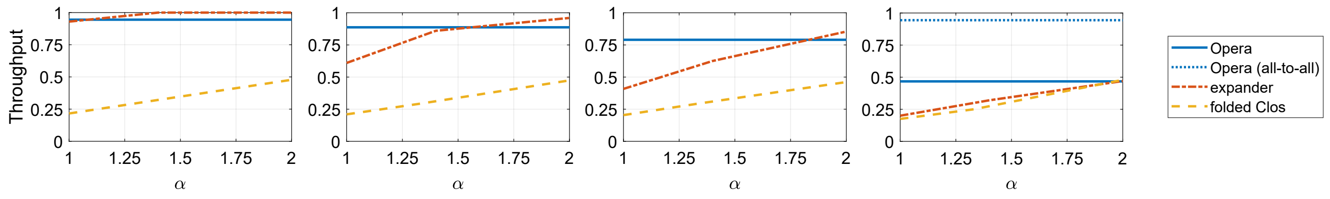


Figure 12: Throughput for (left to right) hotrack, skew[0.1,1], skew[0.2,1], and permutation workloads for $k = 24$ ports.

Opera does not offer an advantage for skewed and permutation workloads when the relative cost of its ports is significantly higher than packet switches ($\alpha > 2$), or in deployments where more than 10% of the link rate is devoted to urgent, delay-intolerant traffic, as described in Section 5.3.

6 Prototype

Priority queueing plays an important role in Opera's design, ensuring that low-latency packets do not get buffered behind bulk packets in the end hosts and switches, and our simulation study reflects this design. In a real system, low-latency packets that arrive at a switch might temporarily buffer behind lower-priority bulk packets that are being transmitted out an egress port. To better understand the impact of this effect on the end-to-end latency of Opera, we built a small-scale hardware prototype.

The prototype consists of eight ToR switches, each with four uplinks connected to one of four emulated circuit switches (the same topology shown in Figure 5). All eight ToR and four circuit switches are implemented as virtual switches within a single physical 6.5-Tb/s Barefoot Tofino switch. We wrote a P4 program to emulate the circuit switches, which forward bulk packets arriving at an ingress port based on a state register, regardless of the destination address of the packet. We connect the virtual ToR switches to the four virtual circuit switches using eight physical 100-Gb/s cables in loopback mode (logically partitioned into 32 10-Gb/s links). Each virtual ToR switch is connected via a cable to one attached end host, which hosts a Mellanox ConnectX-5 NIC. There are eight such end hosts (one per ToR switch) each configured to run at 10 Gb/s.

An attached control server periodically sends a packet to the Tofino's ASIC that updates its state register. After configuring this register, the controller sends RDMA messages to each of the attached hosts, signaling that one of the emulated circuit switches has reconfigured. The end hosts run two processes: an MPI-based shuffle program patterned on the Hadoop workload, and a simple "ping-pong" application that sends low-latency RDMA messages to a randomly selected receiver, which simply returns a response back to the sender. The relatively low sending rate of the ping-pong application did not require us to implement NDP for this traffic.

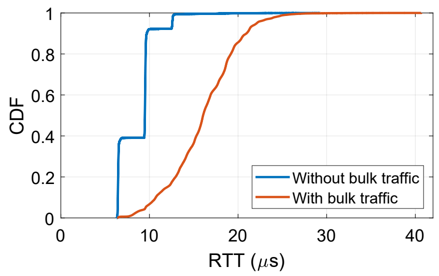


Figure 13: RTT values for low-latency traffic with and without bulk background traffic in the prototype.

6.1 End-to-end latency

Figure 13 shows the observed application-level latency of sending a ping message from a random source to a random destination (and back). We plot this distribution both with and without bulk background traffic. The latency observed without bulk traffic is due to a combination of the path length and the time to forward a packet through Tofino's P4 program, which we observe to be about 3 µs per hop, resulting in latency of up to 9 µs depending on path length. The observed tail is due to RoCE/MPI variance at the end hosts. In the presence of bulk traffic, low-latency packets potentially need to queue behind bulk packets currently being sent from the egress port. Because we emulate circuit switches within the Barefoot switch, each transit of a circuit-switch introduces additional latency that would not be present in a deployment, adding additional latency. For our testbed there are as many as eight serialization points from source to destination, or 16 for each ping-pong exchange. Each serialization point can introduce

#Racks	#Entries	% Utilization
108	12,096	0.7
252	65,268	3.8
520	276,120	16.2
768	600,576	35.3
1008	1,032,192	60.7
1200	1,461,600	85.9

Table 1: Number of entries and resulting resource utilization for Opera rulesets for datacenters of varying sizes.

as much as 1.2 µs (one MTU at 10 Gb/s), or 19.2 µs in total, as shown in Figure 13. The distribution is smooth because when low-latency packets buffer behind bulk packets currently exiting the switch, the amount of remaining time is effectively a random variable.

6.2 Routing state scalability

Opera requires more routing state than a static topology. A straightforward implementation would require the tables in each switch to contain $O(N_{rack})^2$ entries as there are N_{rack} topology slices and $N_{rack}-1$ possible destinations within each slice. We use Barefoot's Capilano compiler tool to measure the size of the ruleset for various datacenter sizes, and compare that size to the capacity of the Tofino 65x100GE switch. The ruleset consists of both bulk and low-latency non-rack-local rules. The resulting number of rules and the percent utilization of the switch's memory are shown in Table 1. Because the practical rulesize limit may be lower than the compiler-predicted size due to hash collisions within the switch, we loaded the generated rules into a physical switch to validate that the rules would fit into the resource constraints. These results show that today's hardware is capable of holding the rules needed to implement Opera, while also leaving spare capacity for additional non-Opera rules.

7 Related work

Opera builds upon previous network designs focused on cluster and low-latency environments. In addition to the folded-Clos and expander graph topologies described thus far, a number of additional static and dynamic network topologies have been proposed for clusters and datacenters.

Static topologies: Dragonfly [30] and SlimFly [8] topologies connect localized pools of high cross-section bandwidth with a sparse inter-cluster set of links, and have been adopted in HPC environments. Diamond [12] and WaveCube [9] statically interconnect switches with optical wavelength MUXes, resulting in a connected topology without reconfiguration. Quartz [32] interconnects switches into rings, and relies on multi-hop forwarding for low-latency traffic.

Dynamic topologies: Several dynamic network topologies have been proposed, which we can group into two categories: those that cannot support low-latency traffic and those that can. In the former case, Helios [16], Mordia [38], and C-Through [44] aim to reactively establish high-bandwidth connections in response to observed traffic patterns; they all rely on a separate packet-switched network to support low-latency traffic. RotorNet [34] relies on deterministic reconfiguration to deliver constant bandwidth between all endpoints, and requires endpoints inject traffic using Valiant load balancing to support skewed traffic. RotorNet requires a separate packet-switched network for low latency traffic.

ProjecToR [20], on the other hand, always maintains a "base mesh" of connected links that can handle low-latency traffic while it opportunistically reconfigures free-space links in response to changes in traffic patterns. The authors initially evaluated the use of a random base network, ruling it out due to poor support of skew. Instead, they proposed a weighted matching of sources and sinks, though it is not clear what the expected diameter of that network would be in general. Similar to ProjecToR, Opera maintains an "always on" base network which consists of a repeating sequence of time-varying expander graphs, which has a well-known structure and performance characteristics.

There are also reconfigurable network proposals that rely on multi-hop indirection to support low-latency traffic. In OSA [10], during reconfiguration some end-to-end paths may not be available, and so some circuit-switch ports can be reserved specifically to ensure connectivity for low-latency traffic. Megaswitch [11] could potentially support low-latency traffic in a similar manner.

8 Conclusions

Static topologies such as oversubscribed folded-Clos and expander graphs support low-latency traffic but have limited overall network bandwidth. Recently proposed dynamic topologies provide high bandwidth, but cannot support low-latency traffic. In this paper, we propose Opera, which is a topology that implements a series of time-varying expander graphs that support low-latency traffic, and when integrated over time, provide direct connections between all endpoints to deliver high throughput to bulk traffic. Opera can deliver a $4\times$ increase in throughput for shuffle workloads and a 60% increase in supported load for skewed datacenter workloads compared to cost-equivalent static networks, all without adversely impacting the flow completion times of short flows.

Acknowledgments

This work is supported by the Department of Energy through grant ARPA-E DE-AR000084, and by the National Science Foundation through grants CNS-1564185, CNS-1553490, CSR-1629973, and CNS-1911104. We would like to thank our shepherd, Manya Ghobadi, as well as the anonymous NSDI reviewers for their useful feedback.

References

[1] IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–300, July 2008.

[2] Mohammad Al-Fares, Alex Loukissas, and Amin Vahdat. A scalable, commodity, data center network architecture. In *Proceedings of the ACM SIGCOMM Conference*, Seattle, WA, August 2008.

[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, April 2010.

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference*, pages 63–74, New Delhi, India, 2010.

[5] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 19–19, San Jose, CA, 2012.

[6] N Alon. Eigen values and expanders. *Combinatorica*, 6(2):83–96, January 1986.

[7] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation*, pages 455–468, Oakland, CA, 2015.

[8] Maciej Besta and Torsten Hoefler. Slim fly: A cost effective low-diameter network topology. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 348–359, New Orleans, Louisana, 2014.

[9] K. Chen, X. Wen, X. Ma, Y. Chen, Y. Xia, C. Hu, and Q. Dong. Wavecube: A scalable, fault-tolerant, high-performance optical data center architecture. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 1903–1911, April 2015.

[10] Kai Chen, Ankit Singlay, Atul Singhz, Kishore Ramachandranz, Lei Xuz, Yueping Zhangz, Xitao Wen, and Yan Chen. OSA: An optical switching architecture for data center networks with unprecedented flexibility. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 18–18, San Jose, CA, 2012.

[11] Li Chen, Kai Chen, Zhonghua Zhu, Minlan Yu, George Porter, Chunming Qiao, and Shan Zhong. Enabling wide-spread communications on optical fabric with MegaSwitch. In *14th USENIX Symposium on Networked Systems Design and Implementation*, pages 577–593, Boston, MA, 2017.

[12] Yong Cui, Shihan Xiao, Xin Wang, Zhenjie Yang, Chao Zhu, Xiangyang Li, Liu Yang, and Ning Ge. Diamond: Nesting the data center network with wireless rings in 3D space. In *13th USENIX Symposium on Networked Systems Design and Implementation*, pages 657–669, Santa Clara, CA, 2016.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, San Francisco, CA, 2004.

[14] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *SIGCOMM Comput. Commun. Rev.*, 36(1):59–62, January 2006.

[15] Facebook. Introducing data center fabric. https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, 2019.

[16] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM Conference*, New Delhi, India, August 2010.

[17] Joseph E. Ford, Yeshayahu Fainman, and Sing H. Lee. Reconfigurable array interconnection by photorefractive correlation. *Appl. Opt.*, 33(23):5363–5377, Aug 1994.

[18] The Apache Software Foundation. Apache Hadoop. https://hadoop.apache.org/, 2018.

[19] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1:1–1:12, Heidelberg, Germany, 2015.

[20] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar,

Madeleine Glick, and Daniel Kilper. ProjecToR: Agile reconfigurable data center interconnect. In *Proceedings of the ACM SIGCOMM Conference*, pages 216–229, Florianopolis, Brazil, 2016.

[21] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, pages 51–62, Barcelona, Spain, 2009.

[22] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, pages 1–14, Oakland, CA, 2015.

[23] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *Proceedings of the ACM Conference on SIGCOMM*, pages 319–330, Chicago, Illinois, USA, 2014.

[24] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, Los Angeles, CA, USA, 2017.

[25] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *BULL. AMER. MATH. SOC.*, 43(4):439–561, 2006.

[26] Ht-sim. The htsim simulator. https://github.com/nets-cs-pub-ro/NDP/wiki/NDP-Simulator, 2018.

[27] Sangeetha Abdu Jyothi, Ankit Singla, P. Brighten Godfrey, and Alexandra Kolla. Measuring and understanding throughput of network topologies. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 65:1–65:12, Salt Lake City, Utah, 2016.

[28] Srikanth Kandula, Jitendra Padhye, and Paramvir Bahl. Flyways to de-congest data center networks. In *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, New York City, NY, October 2009.

[29] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 281–294, Los Angeles, CA, USA, 2017.

[30] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 77–88, Beijing, China, 2008.

[31] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit Switching Under the Radar with REACToR. In *Proceedings of the 11th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–15, Seattle, WA, April 2014.

[32] Yunpeng James Liu, Peter Xiang Gao, Bernard Wong, and Srinivasan Keshav. Quartz: A new design element for low-latency DCNs. In *Proceedings of the ACM Conference on SIGCOMM*, pages 283–294, Chicago, Illinois, USA, 2014.

[33] W. M. Mellette, G. M. Schuster, G. Porter, G. Papen, and J. E. Ford. A scalable, partially configurable optical switch for data center networks. *Journal of Lightwave Technology*, 35(2):136–144, Jan 2017.

[34] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. RotorNet: a scalable, low-complexity, optical datacenter network. In *Proceedings of the ACM SIGCOMM Conference*, Los Angeles, California, August 2017.

[35] William M. Mellette, Alex C. Snoeren, and George Porter. P-FatTree: A multi-channel datacenter network topology. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets-XV)*, Atlanta, GA, November 2016.

[36] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, Budapest, Hungary, 2018.

[37] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, pages 39–50, Barcelona, Spain, 2009.

[38] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang-Chen Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. In *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China, August 2013.

[39] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the ACM SIGCOMM Conference*, London, England, August 2015.

[40] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A lossless network for high-density and disaggregated racks. Technical report, Cornell, https://hdl.handle.net/1813/49647, 2017.

[41] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, pages 183–197, London, United Kingdom, 2015.

[42] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 17–17, San Jose, CA, 2012.

[43] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, pages 205–219, Irvine, California, USA, 2016.

[44] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. c-through: Part-time optics in data centers. In *Proceedings of the ACM SIGCOMM Conference*, pages 327–338, New Delhi, India, 2010.

[45] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y. Zhao, and Haitao Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 443–454, Helsinki, Finland, 2012.

Appendix

A Cost-normalization approach

In this section, we detail the method we used to analyze a range of cost-equivalent network topologies at various network scales and technology cost points. We begin by defining α as the cost of an Opera "port" (consisting of a ToR port, optical transceiver, fiber, and circuit switch port) divided by the cost of a static network "port" (consisting of a ToR port, optical transceiver, and fiber), following [29].

We can also interpret α as the cost of the "core" ports (i.e. upward-facing ToR ports and above) per edge port (i.e. server-facing ToR port). Core ports drive the network cost because they require optical transceivers. Thus, for a folded Clos we can write $\alpha = 2(T-1)/F$ (where T is the number of tiers and F is the oversubscription factor). For a static expander, we can write $\alpha = u/(k-u)$ (where u is the number of ToR uplinks and k is the ToR radix).

We use a $T = 3$ three tier (i.e. three layer) folded Clos as the normalizing basis and keep the packet switch radix (k) and number of hosts (H) constant for each point of network comparison. To determine the number of hosts as a function of k and α, we first solve the for the oversubscription factor as a function of α: $F = 2(T-1)/\alpha$ (note $T = 3$). Then, we find the number of hosts H in a folded Clos as a function of F, k, and α: $H = (4F/(F+1))(k/2)^T$ (note $T = 3$, and F is a function of α). This allows us to compare networks for various values of k and α, but we also estimate α given technology assumptions described below.

Opera's cost hinges largely on the circuit switching technology used. While a wide variety of technologies could be used in principle, using optical rotor switches [34] is likely the most cost-effective because (1) they provide low optical signal attenuation (about 3 dB) [33], and (2) they are compatible with either single mode or multimode signaling by virtue of their imaging-relay-based design [33]. These factors mean that Opera can use the same (cost) multimode or simglemode transceivers used in traditional networks, unlike many other optical network proposals that require expensive and sophisticated telecom grade gear such as wavelength tunable transceivers or optical amplifiers. Based on the cost estimates of commodity components taken from [29] and rotor switch components (summarized in Table 2), we approximate that an Opera port costs about 1.3× more than a static network port (i.e. α=1.3).

B Hybrid cost-performance tradeoff

In Section 5, we evaluated the performance of a hybrid RotorNet which faced one of the six available ToR uplinks to a multi-stage packet switched network (for 1.33× the cost of the other networks evaluated). Here, we consider the tradeoff between FCT and cost for a broader range of hybrid packet switched bandwidths. To consider small fractions of packet switched bandwidth, we allow the bandwidth of a single ToR uplink to be split arbitrarily between the packet and circuit

Component	Static	Opera
SR transceiver	\$80	\$80
Optical fiber (\$0.3/m)	\$45	\$45
ToR port	\$90	\$90
Optical fiber array	-	\$30 †
Optical lenses	-	\$15 †
Beam-steering element	-	\$5 †
Optical mapping	-	\$10 †
Total	\$215	\$275
α ratio	1	1.3

Table 2: Cost per "port" for a static network vs. Opera. A "port" in a static network consists of a packet switch port, optical transceiver, and fiber. A "port" in Opera consists of a packet switched (ToR) port, optical transceiver, and fiber, as well as the components needed to build a rotor switch. The cost of rotor switch components is amortized across the number of ports on a given rotor switch, which can be 100s or 1,000s; we present values in the table assuming 512 port rotor switches. († per duplex fiber port)

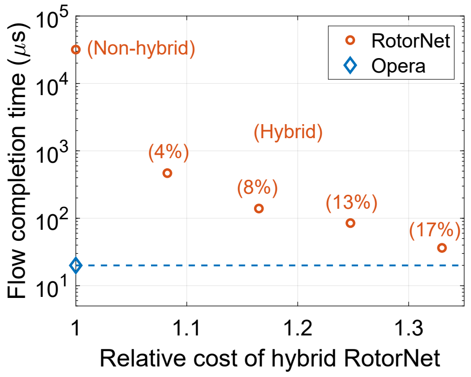


Figure 14: Flow completion times as a function of the cost incurred by adding more packet switched bandwidth to a hybrid RotorNet. The FCTs for Opera are shown for reference. FCTs are shown for 1 kB flows in the Datamining workload running at 25% load. The percent values for hybrid RotorNet indicate the percent of total network bandwidth allocated to the packet switched portion of the network.

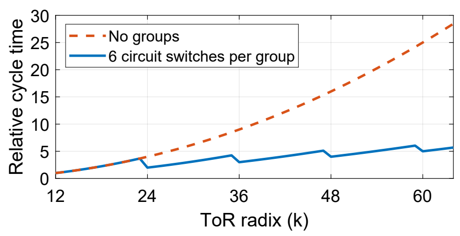


Figure 15: Relative cycle time is improved at larger scale by grouping circuit switches and allowing one switch in each group to reconfigure simultaneously.

networks. Figure 14 shows the resulting tradeoff between cost and the FCTs for 1 kB flows in the Datamining workload running at 25% load (similar trends were observed for other loads and flow sizes). As cost is reduced in hybrid RotorNet (by allocating a smaller percent of total network bandwidth to the packet switched network), FCTs begin to rise substantially due to increased network congestion.

C Reducing cycle time at scale

Larger Opera networks are enabled by higher radix ToR switches, which commensurately increase the number of circuit switches. To prevent the cycle time from scaling quadratically with the ToR radix, we allow multiple circuit switches to reconfigure simultaneously (ensuring that the remaining switches deliver a fully-connected network at all times). As an example, doubling the ToR radix doubles the number of circuit switches, but presents the opportunity to cut the cycle time in half by reconfiguring two circuit switches at a time. This approach offers linear scaling in the cycle time with the ToR radix, as shown in Figure 15. Assuming we divide circuit switches into groups of 6, parallelizing the cycle of each group, the cycle time increases by a factor of 6 from a $k = 12$ (648-host network) to a $k = 64$ (98,304-host network), corresponding to a flow length cutoff for "bulk" flows of 90 MB in the latter case.

D Additional scaling analysis

Figure 16 shows the performance-cost scaling trends for various traffic patterns for networks with $k = 12$ port ToRs. Comparing with Figure 12, we observed nearly identical performance between networks with $k = 12$ and $k = 24$, indicating the (cost-normalized) network performance is nearly independent of scale for all networks considered (folded Clos, static expanders, and Opera).

To analyze this result at a more fundamental level, we evaluated the average and worst-case path lengths for ToR radices between $k = 12$ and $k = 48$ for both Opera and static expanders at various cost points (α). Figure 17 shows that the average path lengths converge for large network sizes (the worst-case path length for all networks including Opera was 4 ToR-to-ToR hops for $k = 24$ and above). Given that the network performance properties of static expanders are

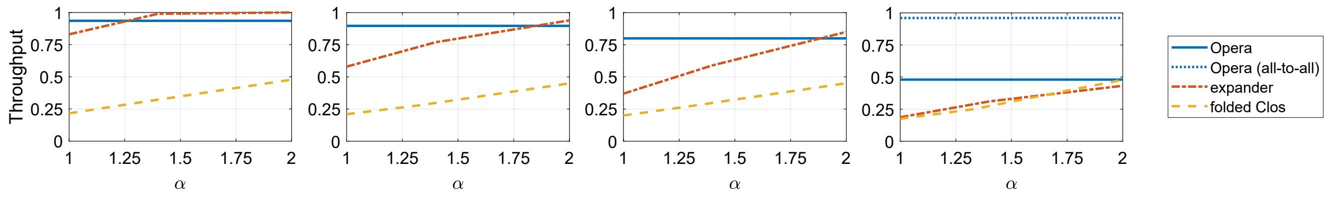


Figure 16: Throughput for (left to right) hotrack, skew[0.1,1], skew[0.2,1], and permutation workloads for $k = 24$ ports.

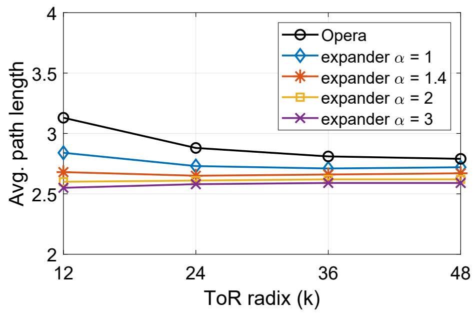


Figure 17: Path lengths for different network sizes (from $k = 12$ with ≈ 650 hosts to $k = 48$ with ≈ 98,000 hosts) and relative cost assumptions (α).

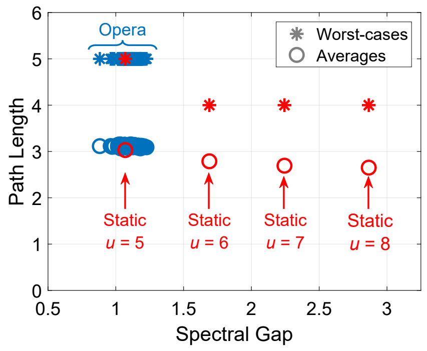


Figure 18: Average and worst-case path lengths and spectral gap for Opera and static expander networks. All networks use $k = 12$-port ToR switches and have between 644 and 650 hosts. Each data point for Opera corresponds to one of its 108 topology slices.

correlated with their path length properties [6], Figure 17 supports our observation that the cost-performance properties of the networks do not change substantially with network size.

E Spectral efficiency and path lengths

The *spectral gap* of a network is a graph-theoretic metric indicating how close a graph is to an optimal Ramanujan expander [25]. Larger spectral gaps imply better expansion. We evaluated the spectral gap for each the 108 topology slices in the example 648-host 108-rack Opera network analyzed in the text, and compared it to the spectral gaps of a number or randomly-generated static expanders with varying d:u ratios. All networks used $k = 12$ radix ToRs and were constrained to have a nearly-equal number of hosts. The results are shown in Figure 18. Note that expanders with larger u require more ToR switches (i.e., cost more) to support the same number of hosts.

Interestingly, when the number of hosts is held constant, we observe that the average and worst-case path length is not a strong function of the spectral gap. Further, we see that Opera comes very close to the best average path length achievable with a static expander, indicating that it makes good use of the ToR uplinks in each topology slice. Opera achieves this good performance despite the fact that we have imposed additional constraints to support bulk traffic with low bandwidth tax: unlike a static expander, Opera must provide a set of $N_{racks} = 108$ expanders across time, and those expanders are constructed from an underlying set of disjoint matchings.

F Additional failure analysis

Opera recomputes paths to route around failed links, ToRs, and circuit switches, and in general these paths will be longer than those under zero failures. Figure 19 shows the correlation between the degree of each type of failure and the average and maximum path length (taken across all topology slices).

For reference, we also analyzed the fault tolerance properties of the 3:1 folded Clos and $u = 7$ expander discussed in the paper. Figure 20 shows the results for the 3:1 Clos and Figure 21 shows results for the $u = 7$ expander. We note that Opera has better fault tolerance properties than the 3:1 folded Clos, but the $u = 7$ expander is better yet. This is not surprising considering the $u = 7$ expander has significantly more links and switches, as well as higher fanout at each ToR.

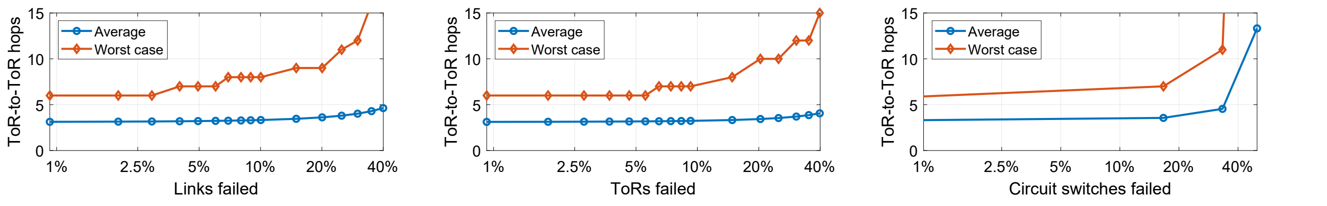


Figure 19: Average and worst-case path length of a 108-rack Opera network with 6 circuit switches and $k = 12$ port ToRs, for various failure conditions. Path length is reported for all finite-length paths. Figure 11 indicates how many ToR-pairs are disconnected (i.e. have infinite path length).

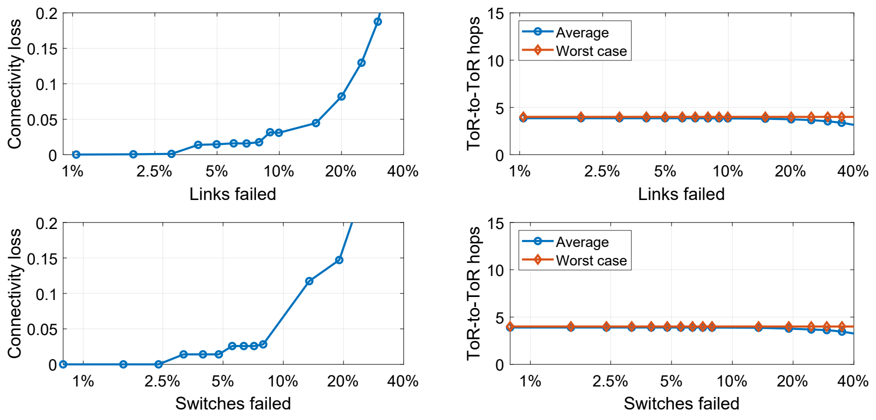


Figure 20: Connectivity loss and impact on path lengths in the 3:1 folded Clos for link failures (top two) and ToR failures (bottom two).

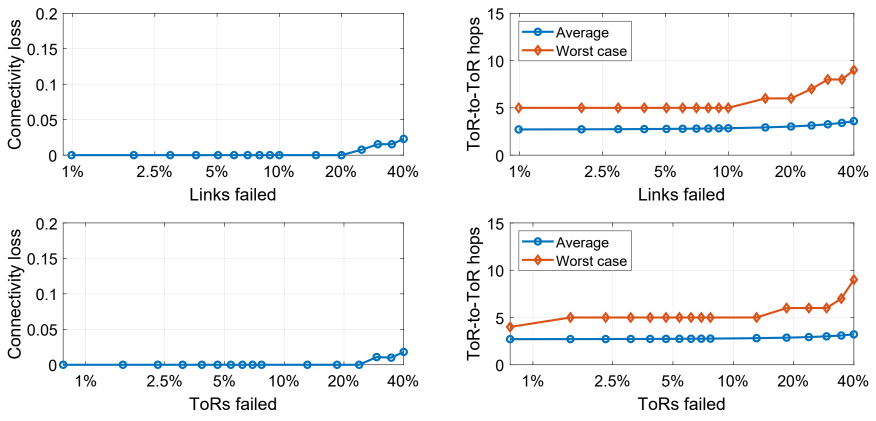


Figure 21: Connectivity loss and impact on path lengths in the $u = 7$ expander for link failures (top two) and ToR failures (bottom two).

Re-architecting Congestion Management in Lossless Ethernet

Wenxue Cheng[†,‡] Kun Qian[†,‡] Wanchun Jiang[§] Tong Zhang[†,‡,*] Fengyuan Ren[†,‡]

[†] *Tsinghua University*

[‡] *Beijing National Research Center for Information Science and Technology (BNRist)*

[§] *Central South University* [*] *Nanjing University of Aeronautics and Astronautics*

Abstract

The lossless Ethernet is attractive for data centers and cluster systems, but various performance issues, such as unfairness, head-of-line blocking and congestion spreading, etc., impede its large-scale deployment in production systems. Through fine-grained experimental observations, we inspect the interactions between flow control and congestion control, and are aware that the radical cause of performance problems is the ineffective elements in the congestion management architecture for lossless Ethernet, including the improper congestion detection mechanism and inadequate rate adjustment law.

Inspired by these insights and findings obtained in experiment investigations, we revise the congestion management architecture, and propose the Photonic Congestion Notification (PCN) scheme, which consists of two basic components: (*i*) a novel congestion detection and identification mechanism to recognize which flows are really responsible for congestion; (*ii*) a receiver-driven rate adjustment method to alleviate congestion in as short as 1 RTT. We implement PCN using DPDK NICs and conduct evaluations using testbed experiments and simulations. The results show that PCN greatly improves performance under concurrent burst workload, and significantly mitigates PFC PAUSE messages and reduces the flow completion time under realistic workload.

1 Introduction

Recently, lossless network has become an attractive trend in data centers and cluster computing systems. Generally, retransmission caused by packet loss readily leads to goodput decrease, completion time increase, and even missing application deadlines [9, 10, 50]. In addition, scaling transport protocols such as Remote Direct Memory Access (RDMA) and Fibre Channel (FC) over data center requires reliable transmission without packet loss due to network congestion [3, 15].

The lossless InfiniBand (IB) [16] is popular in HPC (High performance Computing) cluster systems, but modern data center has already been built with IP/Ethernet technologies that are also dominated in traditional Internet. The data center operators and cloud builders may do some IB, but much less ubiquitous than Ethernet. Furthermore, they are reluctant to simultaneously deploy and manage two separate networks within the same data center [39, 49]. IEEE DCB (Data Center Bridging) [4] is naturally imparted appeal as an enhanced capability of Ethernet, which enables Ethernet to be a consolidated switching fabric that can replace traditionally separated fabrics for special purposes, such as FC for storage, IPC (Interprocess Communication) for HPC, and Ethernet for LAN traffic. Converged Ethernet has significant performance, cost, and management advantages over maintaining separate switching fabrics [8]. To enable lossless semantics for a consolidated Ethernet, both hop-by-hop flow control PFC (Priority-based Flow Control) [6] and end-to-end congestion control QCN (Quantized Congestion Notification) [5] are developed in the link layer to enhance traditional Ethernet. The scalable lossless Ethernet switching fabric is definitely one of the potential candidates for building future data centers to accommodate promising applications, such as RDMA over Converged Ethernet (RoCE) [15], NVMe Over Fabrics [42] and resource disaggregation [23], etc..

Over the last decade, the rise of various Online Data-Intensive (OLDI) applications [31] and virtualized services [40] generate increasingly diverse traffic patterns and specific characteristics, e.g., incast, burst and mixture of mice/elephant flows, etc. [12, 25, 44]. Because it is unclear whether the lossless Ethernet can work effectively in large-scale data centers with such complex traffic, we conduct empirical and experimental investigations to attain the in-depth understanding of congestion management (CM) architecture in lossless Ethernet. The detailed observation and conjoint analysis uncover the radical root of some performance issues, such as congestion spreading and being susceptible to burst traffic. In the light of these insights, we re-architect CM in lossless Ethernet. The key findings and main contributions are summarized as follows.

• Revealing the inadequate elements in existing CM architecture for lossless Ethernet, including: a) The congestion detection mechanism cannot exactly identify congested or uncongested flows when they are churned in the same queue, so that it is unlikely to notify different sources to make discriminative rate adjustments. b) The slow evolution-based rate adjustment of end-to-end congestion control mismatches the fast operations of hop-by-hop flow control.

• Developing a novel CM scheme named Photonic Congestion Notification (PCN), which includes: a) A subtle congestion detection and identification mechanism, which can distinguish real congested flows so as to make a proper rate adjustment for congested or uncongested flows even if they are churned in the same accumulated queue. b) A receiver-driven rate adjustment rule, which can speed up the convergence of rate regulation, and is robust to burst traffic and adaptable to link capacity.

• Implementing PCN using DPDK NICs and conducting evaluations using both testbed experiments and ns-3 simulations. Extensive simulations in the large-scale network with synthesized traffic from real workload show that PCN suppresses PFC PAUSEs by 12%, 47% and 90% compared to QCN, DCQCN and TIMELY respectively, and reduces latency by at most 10x, 11.3x and 13.2x.

2 Background

2.1 Traffic Features in Data Centers

A variety of applications in data centers generate flows with a wide spectrum of traffic patterns and distributions. For example, web search service usually generates short and burst flows. On the other hand, the log file processing introduces few but long-lived flows to transmit bulk of data. Investigations on traffic in many operation data centers show the wide distribution traffic patterns [41]. The size of flows may range from 0.05KB to more than 100MB, and the distribution is quite scattered. Among all traffic, mice flows, which finish sending all packets before receiving any ACK, cannot be adjusted by the end-to-end congestion control scheme. Furthermore, many measurements [18, 19, 33, 41] indicate that the occurrence of mice flow is not only frequent but also bursty. The highly dynamic entering/leaving of mice flows would greatly shock queue length in switches and then the end-to-end latency [12, 13, 44]. Although these flows do not react to the congestion control scheme, they severely disturb the normal operations of the congestion management of switching fabric in data centers or cluster systems.

2.2 Congestion Management in lossless Ethernet

To guarantee losslessness and provide satisfying job completion time under such diverse traffic patterns, congestion management becomes critical and challenging in lossless Ethernet. IEEE DCB [4] specifies a framework for CM consisting of two basic functions, including end-to-end congestion control and hop-by-hop flow control.

The end-to-end congestion control regulates source sending rate actively according to the congestion information reflected by measured variables, such as switch queue length or RTT. Representative solutions include QCN developed by IEEE 802.1 Qau [5], the Layer-3 scheme DCQCN [49], and the RTT-based scheme TIMELY [36]. Although these protocols can constrain the switch queue length and accordingly reduce the packet loss ratio, there is not enough guarantee of zero packet loss. Actually, the uncontrollable burst may be already lost before sources are aware of network congestion, especially when the congestion control loop delay is relatively large or the degree of burst and concurrency is heavy. What is worse, a large number of congestion control mechanisms [5, 21, 27, 36, 38, 49] start flows at the line rate to accelerate the completion of mice flows, which exacerbates the loss problem.

To avoid packet loss due to uncontrollable burst, Priority-based Flow Control (PFC) is defined by IEEE 802.1Qbb [6] to ensure losslessness. With PFC, a switch sends a PAUSE frame to its upstream device (a switch or a NIC) to stop transmission when the ingress queue length exceeds a certain threshold. And a RESUME frame is sent when the queue drains below another threshold. Although PFC can guarantee zero packet loss due to network congestion, it leads to some performance issues such as head-of-line blocking (HLB), unfairness and even deadlock [26, 28, 45, 46, 49]. When PFC is triggered incessantly, the local congestion spreads back to both congested and uncongested sources, and then the network throughput and flow completion time are drastically harmed. The fundamental solution for these performance issues is to eliminate persistent congestion by end-to-end congestion control schemes such that PFC is not triggered incessantly [46, 49].

In total, the end-to-end congestion control needs PFC to prevent packet loss due to transient congestion of uncontrollable burst, and PFC also needs end-to-end congestion control to eliminate persistent congestion. That is, the end-to-end congestion control and hop-by-hop lossless flow control are complementary to each other in lossless Ethernet.

3 Experimental Observation and Insights

3.1 Observations

Although both end-to-end congestion control and hop-by-hop flow control can meet their goals independently under the diverse traffic patterns, their interaction would induce unexpected issues. (1) When burst short flows enter into the network, existing flows in the network would still suffer from the PFC-related side-effects, i.e., congestion spreading and unfairness. (2) After burst leaving the network, congestion control would not efficiently and timely reallocate available bandwidth.

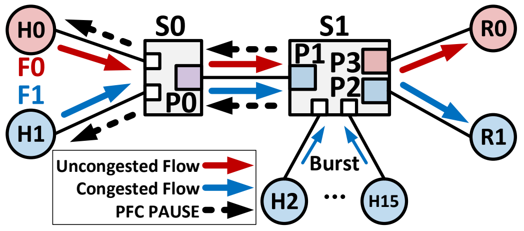
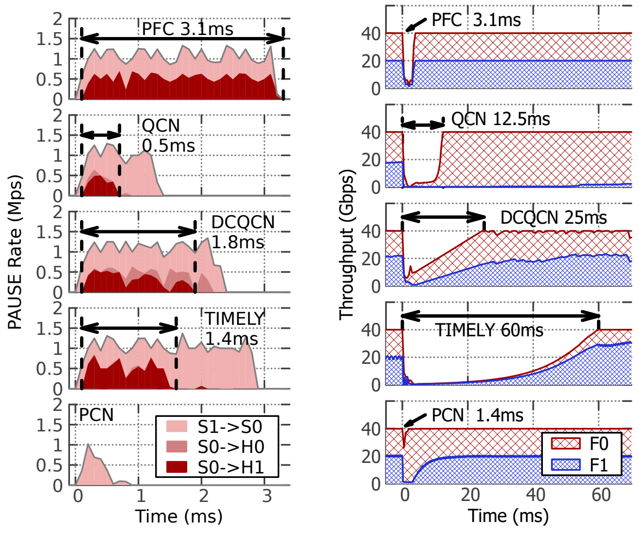


Figure 1: Compact and typical network scenario.

Most existing work considerably concerns the single element in the CM of lossless Ethernet (e.g., congestion control [36, 49]) or special symptoms (e.g., HLB [7], deadlock [28, 29, 45]), but unconsciously overlooks the interaction of congestion control and flow control under diverse traffic patterns, thus is likely to shield the essential cause of aforementioned performance issues. Subsequently, we first conduct a careful, fine-grained and multi-variable observation, and then infer the radical root of special symptoms and issues.

Specifically, we define a compact and typical network scenario, which is not too complex to hinder us capturing the basic principles of both key elements and core mechanisms in the CM of lossless Ethernet. At the same time, it should have sufficiently common features so as to ensure the obtained conclusions and insights are without loss of generality. As shown in Fig.1, we choose a basic unit of a typical network topology in data center, like Clos [22] and Fat-Tree [11], where 16 senders and 2 receivers are connected by two switches. All links are 40Gbps, with a propagation delay of $5\mu s$. The traffic is a mixture of long-lived flows and concurrent burst mice flows. In detail, H0 and H1 start long-lived flows to R0 and R1, respectively. Assume that F0 and F1 achieve their fair bandwidth allocation of the 40Gbps bottleneck link from switch S0 to S1 at the beginning of simulation. At time 0, each sender of H2$\sim$H15 generates 16 short flows to R1 at line rate (i.e., 40Gbps) simultaneously, and the size of each flow is 64KB. Since each mice flow only lasts for $12.8\mu s$ ($<$1 RTT), it is uncontrollable by the end-to-end congestion control mechanisms. These uncontrollable burst flows last for about 3ms in total. We conduct simulations with ns-3 to investigate various CM schemes including PFC, PFC+QCN, PFC+DCQCN, and PFC+TIMELY. All parameters are set to the default values recommended by the related standard [5, 6] and literature [36, 49], and the details are given in § 7. The results are presented in Fig.2.

When PFC is solely employed, the input port P1/S1 pauses its upstream port P0/S0 to avoid packet drops, and the port P2/S1 is congested by concurrent burst flows. Subsequently, "Pause" spreads upstream along with the long flow, and both H0 and H1 are eventually paused. We measure the *PAUSE Rate* (i.e., the rate of transmitting PAUSE messages), and the instantaneous throughput. As shown in Fig.2(a), a congestion tree, which roots from S1, spreads to H0 and H1, appears



(a) Pause rate (b) Throughput

Figure 2: Interactions between PFC and existing congestion controls.

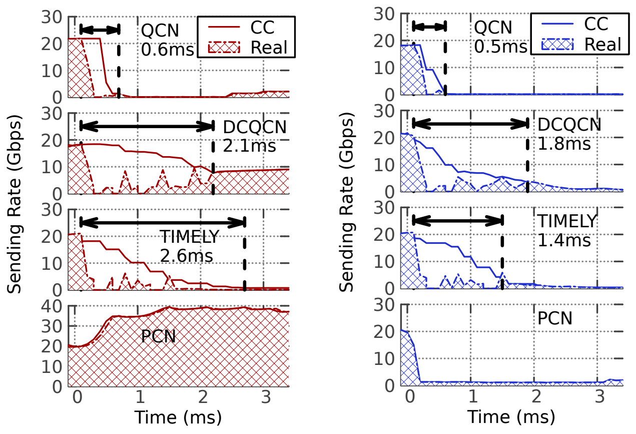
and lasts for 3.1ms (>100 RTT) until the burst mice flows finish. During this process, both the congested flow F1 and uncongested flow F0 face great throughput loss as shown in Fig.2(b), no matter whether they are responsible for the real congestion at port P2/S1.

When QCN, DCQCN or TIMELY works with PFC jointly, the congestion tree still appears, as shown in Fig.2(a). However, its lasting time is reduced to 0.5ms ($\approx$17RTT), 1.8ms ($\approx$57RTT) and 1.4ms ($\approx$47RTT). Surprisingly, the two long flows F0 and F1 may fail to recover to their initial throughput quickly after both concurrent burst flows and congestion tree disappear. As illustrated in Fig.2(b), the throughput loss unexpectedly lasts for 12.5ms with QCN, 25ms with DCQCN and 60ms with TIMELY, respectively, even if the concurrent burst flows last for only 3ms. Totally, the performance of PFC+QCN, PFC+DCQCN and PFC+TIMELY is worse than PFC in this scenario.

3.2 Interaction Issues

To understand the long duration of congestion tree and unexpected great throughput loss, we analyze the dynamic behaviors of flows in detail, and reveal the interaction issues between hop-by-hop flow control and end-to-end congestion controls. We believe that careful analysis and rigorous reasoning from interactive behavior could enlighten us the root causes of various performance issues reported by existing work [26, 34, 37].

1) PFC confuses congestion detection. In above experiments, an ideal end-to-end congestion control scheme should only throttle F1 to 2.5Gbps and allocate flow F0 the remaining



(a) Uncongested flow F0 (b) Congested flow F1

Figure 3: The responsiveness of different congestion controls.

bandwidth (37.5Gbps) of bottleneck link from S0 to S1. However, this ideal bandwidth allocation cannot be achieved even existing congestion controls (QCN, DCQCN and TIMELY) are employed. To explore the cause of this phenomenon, we record the sending rate regulated by the end-to-end congestion control and the real sending rate of the uncongested flow F0 and congested flow F1. The results are presented in Fig.3(a). When congestion tree exists, both the queue length and RTT increase at port P0/S0. Because senders infer congestion according to the feedback information (i.e., queue length or RTT), F0 is also regarded as congested. Hence, the sending rate of F0 is reduced by QCN, DCQCN and TIMELY, even if it contributes nothing to the real congestion point at P2/S1. Therefore, after the congestion tree disappears (as marked by the dash line in Fig.3(a)), the sending rate of F0 is very low although it escapes from the collateral damage of PFC.

In summary, it takes some time for congestion controls to eliminate congestion tree. In this transient process, the large queue length and RTT due to congestion spreading caused by PFC would mislead congestion controls to decrease the sending rate of victim flows (F0 in this example).

2) The slow evolution-based rate adjustment of end-to-end congestion control mismatches the fast hop-by-hop operations of PFC. Fig.3 also unveils the reason why the congestion tree is still created and lasts for tens of RTTs with QCN, DCQCN and TIMELY. Although F0 and F1 are throttled immediately when the concurrent burst mice flows enter, it takes a long time for QCN, DCQCN and TIMELY to reduce the sending rates (the regulation time of different congestion controls are marked in Fig. 3). However, PFC works hop-by-hop and thus the congestion spreads very fast. During the rate decrease of F0 and F1, PFC is triggered incessantly. So the real sending rates of F0 and F1 are mainly determined by PFC rather than end-to-end congestion control, thus the throughput of both F0 and F1 are small. This is why the congestion tree spreading still occurs even if the end-to-end congestion control is employed.

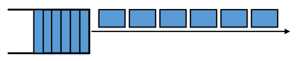
This problem is attributed to the mismatch between the slow evolution-based rate adjustment of end-to-end congestion control and the fast operations of hop-by-hop flow control. More specifically, when the available bandwidth reduces suddenly due to the concurrent burst mice flows, the end-to-end congestion control schemes have no idea of the target rate thus only make rate decrease based on the current sending rate step by step, which is at most 50% per update period. Moreover, the update period is about 20μs (time of transmitting 100 packets) for QCN, 50μs for DCQCN and at least 12.5μs (time of sending 64KB segment) for TIMELY. What's more, when the throughput of F0 and F1 is very small, DCQCN may not receive a single packet in one update period, and would start rate increase automatically. As a result, tens of update periods may be needed to decrease F1's rate to approach the remaining available bandwidth, as shown in Fig.3.

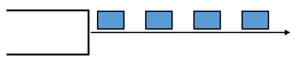
3) The rate increase is inadaptable to dynamic network conditions. After the concurrent burst mice flows vanish and the congestion tree disappears, the sending rates of both F0 and F1 have been throttled, and need to increase step by step. QCN and DCQCN increase the sending rate towards the target rate stored at previous rate decrease in a binary-search way and raise the target rate linearly with a pre-configured value periodically. TIMELY adds the sending rate with a fixed value in each update period. Briefly, all rate increasing methods are linear. Consequently, they fail to take full use of available bandwidth immediately after the disturbance of concurrent burst mice flows. This is why flows F0 and F1 need much longer time to recover to full throughput as presented in Fig.2(b). Moreover, the step of rate increase in each update period needs to be configured adaptively according to network bandwidth. For example, the parameters of QCN, DCQCN and TIMELY tuned for 40Gbps link may be too conservative for 100Gbps link, but too aggressive for 1Gbps link. The tuning of parameters would become difficult in practice.

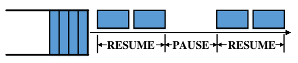
4 Principles

The root cause of all aforementioned performance issues can be concluded as the existing end-to-end congestion control scheme cannot cooperate with hop-by-hop flow control well. To address these issues, we revisit the architecture of CM. We first present a discussion about which elements in existing congestion management introduce these performance issues, and then propose the ways to overcome these incongruities by re-architecting the CM for lossless Ethernet. Briefly, the principles are threefold.

1. The uncongested flow becomes a victim because the existing congestion management cannot identify real congested flows. The operation of PFC would back pressure congestion and contaminate current congestion signals


(a) Real-congestion.


(b) Non-congestion.



(c) Quasi-congestion.

Figure 4: Different states of egress port.

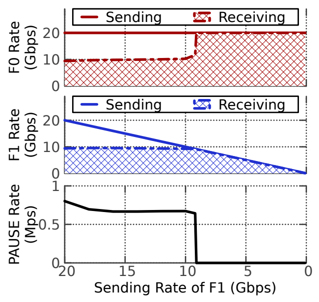


Figure 5: Relationship of PAUSE and receiving rate.

(i.e., queue length and RTT). We need to find out a new mechanism to properly distinguish which flows are really responsible for congestion.

2. Congestion spreading is caused by the slow evolution-based rate decreasing mechanism, thus a fast and accurate rate decreasing solution is indispensable.

3. When burst traffic vanishes, the long-lived flows mainly rely on the linear rate increase to share the released bandwidth, which leads to sluggish convergence and bandwidth waste. Therefore, a prompt rate increasing mechanism should be developed.

4.1 Congestion Detection and Identification

Traditionally, the end-to-end congestion control detects the network congestion based on the measured variables like switch queue length and RTT. However, this congestion detection mechanism is confused by PFC in lossless Ethernet. We need to revise the congestion detection and identification mechanism to avoid this confusion and then correctly identify which flows are really congested.

4.1.1 Detecting Congestion

To aid detecting congestion, we classify the egress ports of switches into the following three states.

Real-Congestion: The ports in real-congestion fully utilize the egress links and the excessive incoming packets accumulate in the buffer, as shown in Fig.4(a). For example, in the previous simulation, when the concurrent burst mice flows start, port P2/S1 is in real-congestion.

Non-Congestion: As illustrated in Fig.4(b), no packets accumulate in the buffer of egress ports, and thus the incoming packet is transmitted immediately. That is, the egress links work normally with utilization less than 100%. The port P3/S1 in above simulation is always in non-congestion.

Quasi-Congestion: The ports in quasi-congestion also keep certain queue length, but the associated egress link is not fully utilized due to PAUSE and RESUME, as depicted in Fig.4(c). Therefore, it is unknown whether the incoming rate of packets exceeds the link capacity or not. For example, in the previous simulation, port P0/S0 turns into quasi-congestion in face of PFC triggers. However, because flows passing through this port would suffer large queue length and delay, the congestion detection mechanism in existing congestion controls (e.g., QCN, DCQCN and TIMELY) dogmatically judges that these flows experience congestion.

Consequently, to distinguish these different states of the egress ports, especially the quasi-congestion state, the impacts of PFC should be taken into consideration when detecting congestion.

4.1.2 Identifying Congested Flows

Owing to the impact of PFC, packets from both congested and uncongested flows are likely to backlog in the same queue length in egress port, which is paused by its downstream ingress port. Therefore, it may be proper to predict potential congestion depending on the queue length of egress port, but indeed unwise to make congestion judgment and provide indiscriminate information to all flow sources, just like QCN and DCQCN. TIMELY also hardly distinguishes whether the flow actually traverses the real congested port by merely measuring RTT and its variations.

To avoid the confused congestion information in existing CM architecture to perturb the normal interaction between flow control and congestion control, and even lead to mutual damage, we advocate decoupling congestion detection and identification functions during re-architecting the CM of lossless Ethernet. The switch is responsible for detecting congestion and providing congestion signals through monitoring the related network status. The end systems synthesize relevant information to judge congestion and identify whether its flow is really congested.

4.2 Receiver-Driven Rate Decrease

The ideal congestion control scheme should throttle the congested flows towards a proper rate directly. To achieve this goal, we need to obtain this proper rate at first. In lossless Ethernet, the proper rate should not trigger PFC but can still keep high throughput. To find this rate, we should answer the following two sub-questions: 1) what is the minimum rate for congested flows to not lose throughput? 2) what is the maximum rate for congested flows to not trigger PFCs?

The first answer is intuitive. It should be the arrival rate of receiver. We define it as *Receiving Rate*. On one hand, the path of congested flows must have at least one real congested port, thus the sum of receiving rates of all flows just achieves the capacity of bottleneck link. On the other hand, if the congested flow decreases rate to less than its receiving rate, there must be idle bandwidth on the bottleneck link, which means that this flow can actually send more data. Thus, the receiving rate is the minimum rate for the congested flows to not lose throughput. The power of receiving rate has also introduced in recent designs like Fast TCP [32], NDP [27] and Homa [38]. They both take advantage of receiving rate to achieve fast convergence when detecting congestion.

Fortunately, the receiving rate is also the answer to the second sub-question. That is, when the sending rate does not exceed the receiving rate, the packets of congested flows do not accumulate at the ingress port of congested switches and then PFCs are not triggered. What's more, this phenomenon occurs regardless of whether the egress port of the same switch is congested or not.

To vividly illustrate this phenomenon, we repeat the experiment in the network scenario given in § 3.1. We start H2∼H4 at line rate to simulate uncontrollable bursts. And both the congestion-irrelevant flow F0 and congestion-relevant flow F1 are controlled by rate limiters. The sending rate of F0 is fixed at its fair allocation (i.e., $20Gbps$), and the sending rate of F1 varies from $20Gbps$ to 0 step by step manually. When the simulation is running, both flows are throttled by their fixed rate limiters and PFC. The sending rates set in rate limiters and receiving rates at the receiver side for these two long flows, as well as the generating rate of PAUSE frames on ingress port p1/S1, are drawn in Fig.5. Obviously, when the sending rate of congestion-relevant flow F1 exceeds 9.5Gbps, its receiving rate is only 9.5Gbps. At the same time, the ingress port p1/S1 generates PFC PAUSEs persistently and the congestion-irrelevant flow F0 is collaterally damaged. On the contrary, when the sending rate of F1 does not exceed 9.5Gbps, no PAUSE frame is generated from port p1/S1 and the congestion-irrelevant flow F0 can achieve its expected throughput. This experiment indicates that throttling congested flows to their receiving rate can prevent more PFC triggers on the associated egress ports, and then suppress congestion spreading in this branch of congestion tree.

Consequently, we obtain a valuable insight, that is **decreasing the rate of congested flows to their receiving rate directly**. It inspires us to design a receiver-driven rate decreasing algorithm to work in harmony with PFC in lossless Ethernet, which will be elaborated in the following.

4.3 Gentle-to-Aggressive Rate Increase

The rate increase should accelerate non-congested flows to rapidly share available bandwidth and then keep at full utilization stably simultaneously. The rate-increase rule of a non-congested flow is needed in two cases.

1) The flow has just turned its state from congested to non-congested. According to our receiver-driven rate decrease principle, the flow rate has reduced to its receiving rate, which implies no PFC trigger and no throughput loss. Thus, the flow has little space for rate increase. Therefore, the rate of this flow should be increased gently.

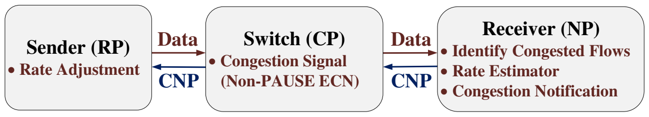

Sender (RP)
• Rate Adjustment
Data
CNP
Switch (CP)
• Congestion Signal (Non-PAUSE ECN)
Data
CNP
Receiver (NP)
• Identify Congested Flows
• Rate Estimator
• Congestion Notification

Figure 6: PCN framework.

2) The flow has remained in the non-congested state for several continuous update periods. In this case, the flow can increase more aggressively to occupy the available bandwidth. Since our receiver-driven rate-decrease rule can sharply reduce the overflowed traffic, the rate increasing mechanism can be designed more aggressively to fulfill network bandwidth quickly.

Therefore, we obtain a suggestion, that is **increasing the rate of non-congested flows gently at first and then aggressively**. It guides us to design a gentle-to-aggressive rate increasing algorithm that can guarantee stability and fast convergence simultaneously.

5 PCN

In this section, based on the principles in § 4, we re-architect congestion management for lossless Ethernet and propose Photonic Congestion Notification (PCN)[1], which is designed to be a rate-based, end-to-end congestion control mechanism to work with PFC in harmony. As shown in Fig.6, PCN is composed of three parts: reaction point (RP), congestion point (CP) and notification point (NP). In general, the CP, which always refers to the congested switch, marks passing packets using a Non-PAUSE ECN (NP-ECN) method to detect whether the egress ports are in real congestion. Notice that a packet marked with NP-ECN does not definitely mean encountering congestion, it requires NP to make the final decision. The NP, i.e., the receiver, identifies the congested flows, calculates their receiving rate and sends the congestion notification packets (CNP) to RP periodically. The RP, which is always the NIC of senders, adjusts the sending rate of each flow according to the information in CNPs. Subsequently, we introduce each part of PCN in details.

5.1 CP Algorithm

We develop the NP-ECN method to detect congestion and generate the congestion signal. The CP algorithm follows the state machine in Fig.7. Suppose that when one egress port of a switch receives a RESUME frame from its downstream

[1]We liken current schemes (e.g. QCN, DCQCN and TIMELY) to quantum, because they can only quantify the network congestion as a whole, but cannot provide different congestion notifications for congested flows and non-congested victim flows, which seems in *quantum entanglement*. And as an analogy, our design is like the photon, which breaks down the entanglement, i.e., directly recognizing the congested flows and allocating them the appreciate rates.

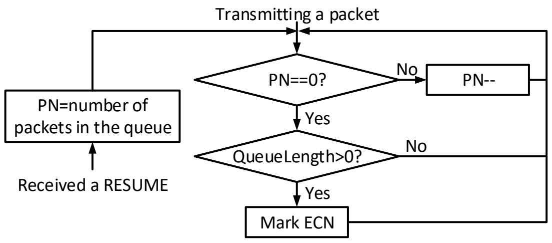
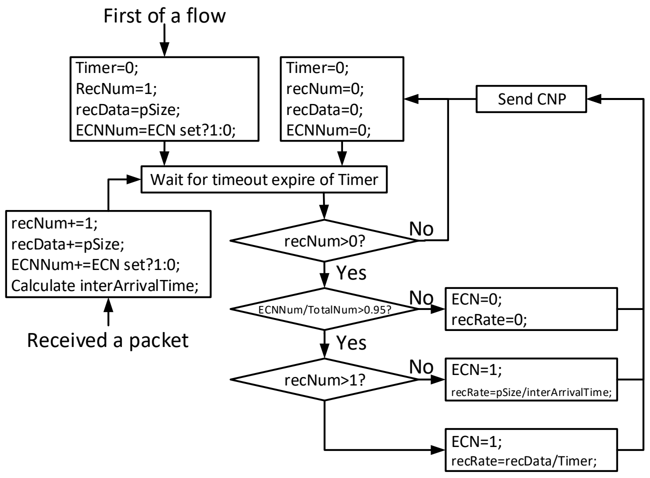


Figure 7: CP state machine.



First of a flow
Timer=0; RecNum=1; recData=pSize; ECNNum=ECN set?1:0;
Timer=0; recNum=0; recData=0; ECNNum=0;
Send CNP
Wait for timeout expire of Timer
recNum+=1; recData+=pSize; ECNNum+=ECN set?1:0; Calculate interArrivalTime;
recNum>0?
No
Yes
ECNNum/TotalNum>0.95?
No
ECN=0; recRate=0;
Received a packet
Yes
recNum>1?
No
ECN=1; recRate=pSize/interArrivalTime;
ECN=1; recRate=recData/Timer;

Figure 8: NP state machine.

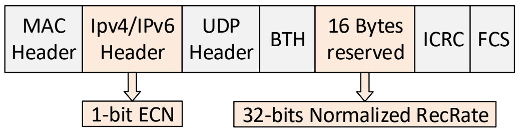


Figure 9: Packet format of CNP

node, there are N packets in the associated waiting queue. NP-ECN will set its counter PN=N. Then the port restarts to transmit packets. Each time a packet is transmitted, the counter is decremented by one, until all of the N paused packets have been transmitted. For these N packets, they will not be marked. And for the later packets that have not been paused and correspondingly PN=0, the switch will mark them with ECN in the traditional method, with the threshold as zero.

In this way, all packets in the real-congestion egress ports will be marked with ECN. On the contrary, the packets are never marked with ECN in the non-congestion ports. And for quasi-congestion ports, the paused packets are not marked with ECN. Meanwhile, when the queue of ingress port is not empty, the packets arriving and leaving the ports in RESUME status are marked with ECN, namely, packets in quasi-congestion ports are partially marked with ECN. In PCN, CP only works for marking a congestion signal on packets and lets the NP node finally determine whether the flow is congested.

It is noted that NP-ECN mechanism can be implemented easily based on the commercial switch equipped with both ECN and PAUSE functions. Compared to the traditional ECN method in commodity switches, the NP-ECN method of PCN requires one more counter per port, and several more lines of logic. The space and computing complexities of modification are both O(1).

5.2 NP Algorithm

The functions of NP include identifying congested flows, estimating receiving rate and sending Congestion Notification Packets (CNP) periodically. T denotes the CNP generation period.

Identifying congested flows: NP identifies the congested flows based on the ECN signal marked by the NP-ECN mechanism. A flow is regarded to be congested if 95% packets received in CNP generation period T are marked with ECN. The value 95% is set empirically to filter some tiny disturbances in practice, such as queue oscillation and priority schedule, which make that one or more packets of real-congested flows are unlikely marked with ECN.

Estimating receiving rate: The receiving rate is calculated directly with T divided by the total size of arrived packets. Noticeably, the receiving rate of flow may be so small that just one packet arrives in several CNP generation periods. To address this special case, PCN also records the inter-arrival time of packets at the NP. When the inter-arrival time is larger than T, NP estimates the receiving rate by replacing T by the inter-arrival time.

Generating CNPs: The NP sends CNPs to notify the source of flow with the receiving rate in period T, which is set to be $50\mu s$ similar to DCQCN. Moreover, PCN generates CNP explicitly when the flow needs either rate-decrease or rate-increase, different from DCQCN which only generates CNPs to notify rate-decrease. And the CNP is not generated when none of its packets is received in period T. In details, the format of CNP packets is compatible with the CNP packet in RoCEv2 [15], as shown in Fig.9. The main information encapsulated by CNP includes 1-bit ECN in the IPv4/IPv6 header and 32-bit RecRate in the reserved segment, which carries the receiving rate normalized by $1Mbps$. The state machine of NP algorithm is summarized in Fig.8.

5.3 RP Algorithm

Algorithm 1 describes the pseudo code of how RP adjusts the sending rate according to the information in CNP. In the beginning, flows start at the line rate to improve flow completion time (FCT) for short flows.

Rate Decrease: When RP receives a CNP with ECN-marked, it conducts a rate decrease following the rule in line 6. Instead of resetting the sending rate to the receiving rate di-

Algorithm 1 PCN RP Algorithm.

```
1: $sendRate \leftarrow lineRate$
2: $w \leftarrow w_{min}$
3: repeat per CNP ($CE$, $recRate$)
4:   if $CE == 1$ then
5:     (CNP notifies rate decrease)
6:     $sendRate \leftarrow \min\{sendRate,\ recRate \cdot (1 - w_{\min})\}$
7:     $w \leftarrow w_{\min}$
8:   else
9:     (CNP notifies rate increase)
10:    $sendRate \leftarrow sendRate \cdot (1 - w) + lineRate \cdot w$
11:    $w \leftarrow w \cdot (1 - w) + w_{\max} \cdot w$
12: until End of flow
```

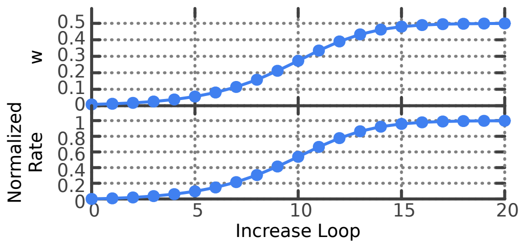


Figure 10: Evolution of w and rate when a sender receives continuous CNP notifying rate increase.

rectly as discussed in § 4.2, a small discount $w_{\min}$ is conducted, such that the build-up queue in switches can be drained out. Accordingly, during draining the build-up queue, the *recRate* may be larger than the *sendRate*, the sending rate should be non-increasing in line 6.

Rate Increase: When RP receives a CNP without ECN-marking, it makes rate adjustments following the law in line 10 and 11. Specifically, the RP increases the sending rate by computing a weighted average of its current value and the line rate. This rate increase law is effective in multiple folds.

(1) The ideal sending rate can be reached as it always stays between the current sending rate and the line rate.

(2) Since the value of w is identical for all flows, the slow flows increase more aggressively than fast flows, which is beneficial to fairness.

(3) The weight w changes automatically from the minimum value $w_{\min}$ to the maximum value $w_{\max}$ such that PCN can realize the gentle-to-aggressive rate increase as discussed in § 4.3. For example, when $w_{\min} = 1/128$, $w_{\max} = 0.5$, and CNPs without ECN-marking are received successively, the evolution of w and the sending rate from 0 to the lineRate are presented in Fig.10. The sending rate grows by no more than 10% of the line rate in the first 5 CNPs, but increases to 95% of the line rate after only 15 CNPs.

(4) Any parameter configurations are not specially required to adapt to the upgrade of link capacity from 1Gbps to even 400Gbps.

5.4 Discussion

As discussed in §4, the main root of performance issues in current lossless Ethernet is the improper interaction between PFC and end-to-end congestion control schemes. We demonstrate that PCN solves the core problems in lossless Ethernet using a minimal implementation cost.

Implementation requirement: To implement PCN, a little switch modification is needed. Compared to the traditional ECN method in commodity switches, the NP-ECN method of PCN (see Fig.7) only requires one more counter per port, and several more lines of logic. The space and computing complexities of modification are both O(1).

Benefits: To demonstrate the advantages of PCN, we enable PCN and repeat the simulations in § 3.1, and the results are also inserted into Fig.2 and Fig.3, respectively. The results in Fig.2(a) tell that PAUSE in both S0->H0 and S0->H1 links are completely avoided and only a handful of PAUSE fleetingly appears in the S1->S0 link, but congestion spreading is quickly suppressed and congestion tree is not generated. The results in Fig.3 confirm that PCN can help the uncongested flows grab idle bandwidth quickly, and regulate the congested flows to proper rates correctly and promptly. PCN increases F0 to fully utilize network bandwidth during concurrent bursts. After the concurrent burst vanishes, F0 and F1 fairly share bandwidth without wasting network resources or triggering PFC PAUSEs as shown in Fig.2(b).

6 Theoretical Analysis and Parameter Setting

6.1 Theoretical Analysis

We build a fluid model of PCN and analyze its performance, including convergence, fairness, and stability. The main conclusions are summarized in the following propositions and the detailed analyses are listed in Appendix A.

Proposition 1. *PCN can achieve convergence of total rate towards the bottleneck capacity as fast as in only one control loop, i.e., one RTT.*

Proposition 2. *PCN can always fairly share the bottleneck link, i.e., $R_i \rightarrow \frac{C}{N}$ regardless of the initial sending rates and parameter settings, where R_i is the receiving rate of flow i, N is the number of sources sharing the bottleneck link, and C is the link capacity.*

Proposition 3. *PCN is stable and the oscillation of both the queue length and rate are bounded in the steady state. The maximum oscillation ranges of queue length (ΔQ) and receiving rate of flow i (ΔR_i) are*

$$\Delta Q = (N - 2 + w_{\min}) w_{\min} CT \tag{1}$$

$$\Delta R_i \rightarrow w_{\min} C \tag{2}$$

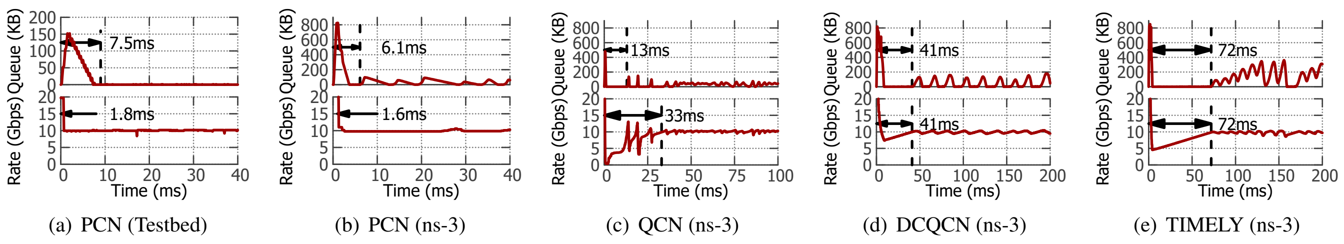


Figure 11: Dynamic behavior of PCN, DCQCN and TIMELY.

6.2 Parameter Settings

Based on the above conclusions, we can obtain some practical guidelines towards parameter settings, including the CNP generating period T, and the minimum and maximum value of weight factor w.

CNP generating period T: It should be identical for all flows. It is noteworthy that T is also the control loop period, thus a large T will damage the responsiveness of PCN. However, in practice, there exists an inherent control loop delay, i.e., RTT. If T is smaller than RTT, PCN is hardly aware of status change in the last control loop, which leads to overmuch adjustments and considerable oscillations. Therefore, the recommended T should be the maximum value of RTT in networks, which can be estimated in advance.

Minimum weight $w_{\min}$: The value of $w_{\min}$ should make a trade-off between fast convergence and stability. A large/small $w_{\min}$ will speed up/slow down the convergence of queue length, but make the flow oscillate more/less aggressively at steady state. According to Proposition 2, Equation (1) and (2), we recommend the proper value of w to be $0.1/N$, which limits the aggregate rate oscillation not exceeding $0.1C$ and the queue oscillation less than $0.1CT$.

Maximum weight $w_{\max}$: The value of $w_{\max}$ determines how aggressively a flow increases when the network is detected under-utilized continuously. Thus an aggressive $w_{\max}$ is recommended, i.e., $w_{\max} = 0.5$.

7 Evaluation

We evaluate the performance of PCN in a variety of settings using testbed experiments (§ 7.1) and ns-3 simulations (§ 7.1 $\sim$ 7.4), and compare it against QCN, DCQCN and TIMELY. The functional modules of our simulator are developed based on the open project for DCQCN [48] and code snippet (per-packet pacing version) for TIMELY [35], and all parameters are set to the default values recommended by the related literatures [36,49]. All experiments enable PFC with $X_{OFF} = 512KB$.

7.1 Basic Properties

In this subsection, we verify the basic function of PCN using simple synthetic microbenchmarks.

Testbed setup: Since current commodity switches do not provide the interface to modify the ECN-marking logic, we implement PCN upon DPDK [1]. We plug two Intel 82599 NICs to one PowerEdge R530 server to act as PCN's CP. Each NIC has two 10Gbps Ethernet ports and the server is equipped with dual Intel Xeon E5-2620 v3 CPUs (6 cores, 2.4GHz). Thus, the server can work as a four-port switch. By deploying DPDK in the server, both PFC and NP-ECN are implemented based on the reference test-pipeline project [2].

DPDK also enables the implementation of our special packet processing required at NICs. On the sender side, the rate limiter at a per-packet granularity is employed for rate adjustment. On the receiver side, PCN receives packets, records ECN marking information, and sends back CNP packets periodically.

Scenario: We use a dumbbell topology where 3 pairs of sender and receiver share the same 10Gbps bottleneck link. Specially, the number of flows on one of three pairs is twice of that on other two. We run this experiment on both hardware testbed and ns-3 simulator for cross-validation. In both testbed experiments and simulations, the RTT is measured to be about $500\mu s$, thus the same configuration is kept in simulations.

Fine-grained observation: First, four long-lived flows are launched and the dynamic behaviors of PCN, QCN, DCQCN and TIMELY is observed. The evolutions of queue length in bottleneck and the aggregate sending rate are depicted in Fig.11. As illustrated in Fig.11(a) and 11(b), PCN exhibits the same performance on the testbed and simulator. Comparing Fig.11(b) to 11(c), 11(d) and 11(e), PCN outperforms QCN, DCQCN and TIMELY in terms of fast convergence and stability.

In both testbed experiment and simulation, PCN regulates the aggregate sending rate to the bottleneck capacity within 2ms (4 RTT), which is 20x, 25x and 45x faster than that with QCN, DCQCN and TIMELY, which is benefited from the receiver-driven rate-decrease method of PCN. It can throttle the incoming traffic to match the bottleneck capacity directly, rather than explore the available bandwidth round by round. Consequently, PCN can limit the bottleneck queue length

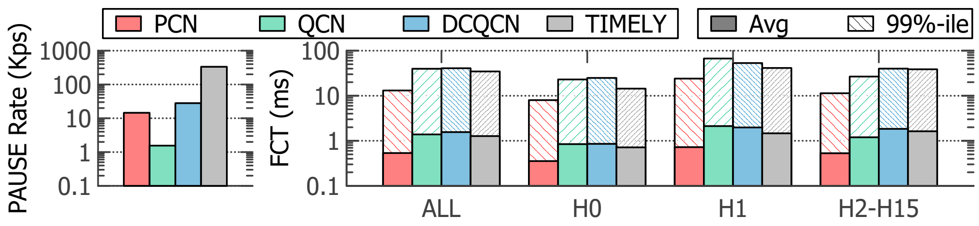


Figure 12: Generating rate of PAUSEs and FCT under concurrent burst.

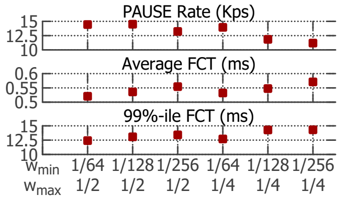


Figure 13: Performance of PCN with different w_{min} and w_{max}.

at a very low level (about several packets) in no more than 7.5ms (∼15 RTT), while it costs 13ms (∼26 RTT) for QCN, 41ms (∼80RTT) for DCQCN and 72ms (∼144 RTT) for TIMELY.

In the steady state, PCN oscillates with low amplitude in both testbed experiment and simulation. The queue length almost approaches zero and the aggregate sending rate keeps near 10Gbps. QCN has the similar performance. However, both DCQCN and TIMELY lead to large oscillations and high buffer occupancy. This advantage comes from the congestion detection method of PCN. The threshold of queue length for ECN marking is set to zero, rather than a positive value.

7.2 Burst Tolerance

One advantage of PCN is robustness against PFC triggers caused by concurrent burst flows. Next, we use the basic scenario in Fig.1 to evaluate PCN in the typical head-of-line scenario. All links are 40Gbps with $5\mu s$ propagation delay, hosts $H0 \sim H15$ generate flows according to the heavy-tailed Hadoop workload [44] with exponentially distributed inter-arrival time. Specially, the workload generators at hosts $H2 \sim H15$ are set to be synchronous to simulate concurrent bursts. The target load at the two bottleneck links is set to 0.6. We measure the pause rate and flow completion time (FCT) of PCN and compare them with QCN, DCQCN and TIMELY.

The left subgraph in Fig.12 shows the generating rate of PFC PAUSEs. QCN triggers the smallest PAUSEs, and PCN can prevent at least 53% and 92% of PFC PAUSEs compared to DCQCN and TIMELY, respectively. And the average and 99_{th} percentile FCTs from different hosts are drawn in the right subgraph in Fig.12. The solid bar at the bottom indicates the average FCT and the upper stripe bar shows the 99_{th} percentile value. Clearly, PCN performs better than QCN, DCQCN and TIMELY for all kinds of hosts.

1) Actually, QCN avoids PAUSEs by drastically reducing the sending rate, which likely leads to poor link utilization and high FCT for long-lived flows. On the contrary, PCN can prevent PAUSEs without harming throughput, and then achieves 2.25x∼3.03x shorter FCT than QCN.

2) For the victim host H0, PCN achieves 2.4x and 2.0x faster average FCT compared to DCQCN and TIMELY, which is mainly benefited from a fact that PCN can mitigate PFC triggers between two switches. For the concurrent burst from $H2 \sim H15$, PCN can keep the buffer at egress port P2 nearly empty, and thus obtain an improvement of 3.5x and 3.4x in the 99_{th} percentile FCT compared to DCQCN and TIMELY. And for host H1, whose flows traverse two congested switches, the flow transmission speed of PCN is at least 2.2x of DCQCN and 1.7x of TIMELY.

Flow size	% of number		% of traffic	
	W1	W2	W1	W2
0KB-10KB (S)	80.14	70.79	3.08	0.22
10KB-100KB (M)	10.32	16.59	5.89	1.56
100KB-1MB (L)	9.12	3.52	83.8	1.53
1MB- (XL)	0.41	9.1	7.04	96.7

W1	Web-server rack at Facebook [44].
W2	Hadoop cluster at Facebook [44].

Table 1: Flow size distribution of realistic workloads.

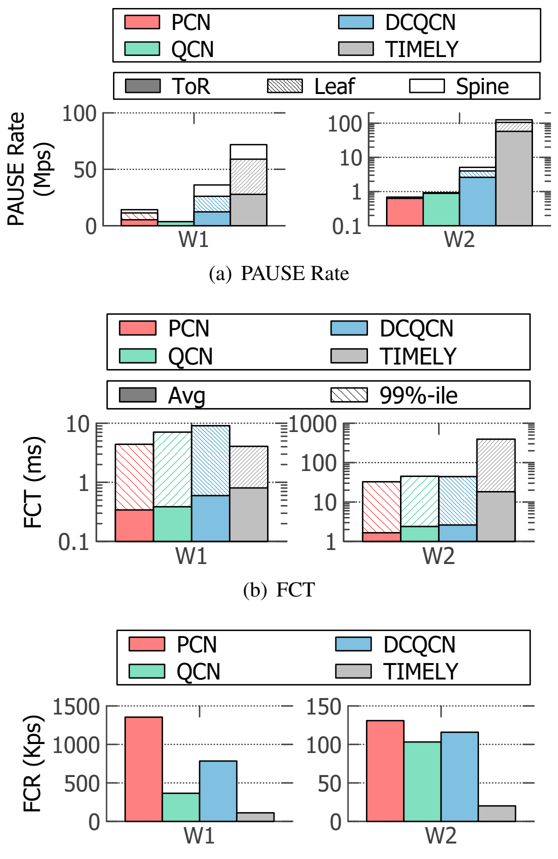
7.3 Parameter sensitivity

As discussed in § 6.2, the minimal and maximal of weight factor w_{min} and w_{max} determine the convergence speed and oscillation amplitude in steady state. To evaluate the parameter sensitivity, we repeat the concurrent burst simulation with different w_{min} and w_{max} values. Fig.13 shows the result. With the changes of w_{min} and w_{max}, PCN can always achieve the satisfied performance. As w_{max} decreases, switch S0 receives fewer PFC PAUSEs from its downstream device, but the 99%-tile of FCT grows a little. Meanwhile, the value of w_{min} has almost no impact on pause rate, but the small w_{min} increases FCT slightly. The results indicate that our recommended parameter settings are proper.

7.4 Realistic Workloads

In this subsection, we evaluate the performance of PCN with realistic workload.

Scenario: We consider an 8-pod clos network. Each pod consists of 2 Leafs, 4 ToRs, and 64 hosts, and communicates with other pods through 8 spines. The link capability is 10Gbps below ToRs and 40Gbps above them, and the link delay is $5\mu s$. The over-subscription ratio is 1:1 at the ToR switch layer, so does in other layers. To support multi-path



(a) PAUSE Rate

(b) FCT

(c) FCR

Figure 14: Performance for realistic workloads.

capability, Equal Cost Multi Path (ECMP) routing scheme is used. In this configuration, the congestion tends to occur at the last hop. When PFC is employed to guarantee losslessness, the root congestion at the last hop may spread to the whole network.

Workloads: We choose two different workloads, whose flow size distribution is listed in Table 1. These two workloads are typical traffic pattern in operation data centers: (1) most flows are short, and (2) most traffic is constituted by few but large flows. The difference is that W2 contains more heavy-tailed flows.

We generate over 50 thousands of flows with exponentially distributed inter-arrival time, and configure the target load at 0.6 for ToR down-links. The source and destination of each flow are arbitrarily selected with a random in-cast ratio ranging from 1 to 15.

Fig.14 presents the results. The generating rates of PFC PAUSEs from different switch layers are drawn in Fig.14(a), where the solid bat at the bottom indicates the PAUSE rate from the ToRs, the middle stripe bar denotes that from the Leafs, and the top empty bar shows that from the Spines. In Fig.14(b), we draw the statistical FCT of all flows, and Fig.14(c) shows the flow completion rate (FCR) i.e., the number of completed flows per second. Subsequently, we compare the performance of PCN with QCN, DCQCN and TIMELY under different workloads.

(1) W1 contains the most S size flows in number and the most L size flows in bytes. Under this workload, the network congestion condition would change dramatically. Although PCN triggers 5.73x more PFC PAUSEs than QCN, it achieves 1.60x faster 99%-ile FCT and 3.70x larger FCR. This is because QCN reduces the sending rate of large flows so drastically that the network becomes seriously underload. Since PCN can rapidly detect the congestion point and adjust the rate of congested flows, short flows experience a low queuing delay and complete quickly. This can improve the overall FCT and increase FCR. Compared with DCQCN and TIMELY, PCN avoids 64% and 75% PFC PAUSEs, speeds up 1.75x and 2.35x in average FCT, and obtains 1.73x and 12.16x FCR, respectively.

(2) W2 is significantly heavy-tailed, where the S size flows occupy almost 80% of the number and less than 1% of the bytes, while the XL size flows only take less than 10% of the number but occupy almost all bytes. Under this workload, PCN suppresses 35%/89%/99% PAUSEs, speeds up 1.44x/1.57x/10.96x in average FCT and achieves 1.27x/1.13x/6.5x more FCR compared with QCN, DCQCN and TIMELY, respectively.

7.5 External Evaluations

Furthermore, we conduct external evaluations to explore PCN's performance in more scenarios. The detailed descriptions are in Appendix B, and the main findings are five folds.

Flow Scalability: PCN can hold as many as 1024 concurrent long flows, guaranteeing few PFC PAUSEs, low and stable queue length, near-full network utilization, as well as good fairness.

Adversarial Traffic: When facing dynamic flows entering and exiting with an interval of 10∼100 control loops, the end-to-end congestion control schemes fail to start the fast rate increasing algorithm. Compared with QCN, DCQCN and TIMELY, PCN can alleviate but not fully eliminate the interruption from adversarial traffic.

Multiple Bottlenecks: In the parking lot scenario with N bottlenecks, PCN allocates bandwidth following *proportional fairness*. That is, it allocates $\frac{1}{N+1}$ of capacity to the flow that passes all N bottlenecks.

Multiple Priorities: When concurrent burst in higher priority leads to severe oscillation of available bandwidth in lower priority, PCN triggers less PAUSE compared with DCQCN and TIMELY. Consequently, PCN outperforms other schemes in speeding up the overall flow completion.

Deadlock: PCN can not essentially prevent PFC deadlock, neither can other end-to-end congestion control schemes, but can significantly decrease the probability of deadlock. Compared to DCQCN and TIMELY, PCN can reduce 79.2% and 96.7% of deadlocks, respectively.

8 Related Work

The lossless switching fabric is a lasting topic. Here we only present a brief survey on the related work of lossless Ethernet and its congestion management, as well as receiver-driven rate control schemes.

Scaling RDMA over data centers. There are two lines in scaling RDMA over data centers. The first line, such as DCB [4] and RoCE [15, 26], attempts to enhance Ethernet with lossless property using PFC. It requires little modification to the well-tested RDMA transport stack but involves new issues caused by PFC. So an appropriate end-to-end congestion control scheme is needed. And the second line, such as Resilient RoCE [34] and IRN [37], tries to improve the RDMA transport stack to tolerate packet loss. Thus it can scale RDMA over lossy networks. We prefer the first line. We think the lossless Ethernet is more potential. On one hand, not just for RDMA, lossless Ethernet makes it easier to enable various well-tested transport protocols in data centers. It does not require NICs to support selective retransmission using the limited storage resources. On the other hand, lossless Ethernet can avoid retransmission of lost packets, and then can improve both network latency and throughput performance.

Lossless Ethernet switching fabric. It is always attractive to build cost-effective, efficient and large-scale lossless switching fabric leveraging commodity Ethernet chips. The related studies broadly follow three fundamental ways, which are reservation, explicit allocation, and congestion management. TDMA Ethernet [47] advocates reserving slots by deploying TDMA MAC layer. Fastpass [43] conducts explicit bandwidth allocations by a centralized arbiter to determine the time at which each packet should be transmitted and the path it should take. Whether TDMA Ethernet or Fastpass, they leverage non-conflict bandwidth access to build lossless Ethernet. However, due to slot wastage and unneglectable signal overheads, their flexibility and scalability in large-scale and ultra-high speed networks need to be further validated in practice. The third approach is to enhance traditional lossy Ethernet by introducing congestion management.

Congestion management for lossless Ethernet. IEEE DCB task group [4] defines the congestion management framework and develops concrete mechanisms, including PFC [6] and QCN [5], to enhance traditional Ethernet to be Converged Ethernet where losslessness should naturally be indispensable. To enable RoCE deployment in large-scale IP-routed data center networks, DCQCN [49] is developed through replacing the congestion notification mechanism defined in QCN with ECN in Layer 3, and then stitching together pieces of rate adjusting laws from QCN [5] and DCTCP [12]. TIMELY [36] follows the implicit congestion detection mechanism developed by TCP Vegas [20] and uses delay measurements to detect congestion, and then adjusts transmission rates according to RTT gradients. Both explicit and implicit congestion detection mechanisms in existing end-to-end congestion control schemes cannot identify the real congested flows, thus the performance issues in lossless Ethernet, such as HoL, congestion spreading and unfairness, are hardly solved essentially. In addition, IEEE 802.1 pQcz [7] has been supplemented to prevent PFC harming victim flows by isolating congestion. However, modification of current commodity switches is required to add more functions. In comparison, the congestion detecting mechanism in our PCN can correctly identify congested flows, moreover is practicable and back-compatible, which endows fundamental advantages for congestion management in lossless Ethernet.

Receiver-driven rate control. Recently, a series of receiver-driven rate control schemes have been proposed, such as ExpressPass [21], NDP [27] and Homa [38]. ExpressPass proactively controls congestion even before sending data packets by shaping the flow of credit packets in receivers. Both NDP and Homa also use the receiver-driven method to allocate priority to different flows in lossy data center networks. The receiver-driven rate adjustment in our PCN not only has the similar benefit of matching the incoming traffic load to the network capacity in one RTT, but also can drastically mitigate PFC triggers in one RTT as well, which is especially appropriate for lossless Ethernet.

9 Conclusion

This paper re-architects congestion management for lossless Ethernet, and proposes Photonic Congestion Notification (PCN), which is appropriate for lossless Ethernet by two ingenious designs: (*i*) a novel congestion detection and identification mechanism to recognize which flows are really responsible for congestion; (*ii*) a receiver-driven rate adjustment scheme to alleviate congestion in as fast as one loop control round, i.e., one RTT. PCN can be easily implemented on commodity switches with a little modification. Extensive experiments and simulations confirm that PCN greatly improves performance, and significantly mitigates PFC PAUSE messages and reduces the flow completion time under realistic workload.

Acknowledgments

The authors gratefully acknowledge the shepherd Rachit Agarwal and the anonymous reviewers for their constructive comments. This work is supported in part by the National Key Research and Development Program of China (No.2018YFB1700103), and by National Natural Science Foundation of China (NSFC) under Grant 61872208.

References

[1] DPDK. http://dpdk.org/.

[2] DPDK test-pipeline. https://github.com/DPDK/dpdk/tree/master/test/test-pipeline.

[3] FCoE (Fibre Channel over Ethernet). http://fcoe.com/.

[4] IEEE 802.1 Data Center Bridging Task Group. http://ieee802.org/1/pages/dcbridges.html.

[5] IEEE 802.1 Qau - Congestion Notification. https://1.ieee802.org/dcb/802-1qau/.

[6] IEEE 802.1 Qbb - Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/.

[7] IEEE P802.1 Qcz - Congestion Isolation. https://1.ieee802.org/tsn/802-1qcz/.

[8] Benefits of Storage Area Network/Local Area Network (SAN/LAN) Convergence, 2009. http://i.dell.com/sites/doccontent/business/smb/sb360/en/Documents/benefits-san-lan-convergence.pdf.

[9] Monitoring and troubleshooting, one engineer's rant, 2011. https://www.nanog.org/meetings/nanog53/presentations/Monday/Hoose.pdf.

[10] Keeping cloud-scale networks healthy, 2016. https://video.mtgsf.com/video/4f277939-73f5-4ce8-aba1-3da70ec19345.

[11] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[12] Mohammad Alizadeh, Albert G Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *ACM SIGCOMM*, 2010.

[13] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *In Proceedings of USENIX NSDI*, 2012.

[14] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick Mckeown, Balaji Prabhakar, and Scott Shenker. pfabric: minimal near-optimal datacenter transport. In *ACM SIGCOMM*, 2013.

[15] InfiniBand Trade Association. Infiniband Architecture Specification Volume 1 Release 1.2.1 Annex A17: RoCEv2, 2014. https://cw.infinibandta.org/document/dl/7781.

[16] InfiniBand Trade Association. Infiniband Architecture Specification Volume 2 Release 1.3.1. 2016. https://cw.infinibandta.org/document/dl/8125.

[17] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *In Proceedings of USENIX NSDI*, 2015.

[18] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[19] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. In *Proc. of the 1st ACM workshop on Research on enterprise networking*, pages 65–72. ACM, 2009.

[20] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[21] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proc. of ACM SIGCOMM*, 2017.

[22] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[23] Peter Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *In Proceedings of USENIX NSDI*, 2016.

[24] Peter Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: distributed near-optimal datacenter transport over commodity network fabric. In *ACM CoNEXT*, 2015.

[25] Albert G Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM*, 2011.

[26] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *ACM SIGCOMM*, 2016.

[27] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wojcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*, 2017.

[28] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *ACM HotNets*, 2016.

[29] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical pfc deadlock prevention in data center networks. In *ACM CoNEXT*, 2017.

[30] Raj Jain, Arjan Durresi, and Gojko Babic. Throughput fairness index: An explanation. 1999.

[31] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. 2013.

[32] Cheng Jin, David X Wei, and Steven H Low. Fast tcp: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, volume 4, pages 2490–2501. IEEE, 2004.

[33] Rishi Kapoor, Alex C Snoeren, Geoffrey M Voelker, and George Porter. Bullet trains: a study of nic burst behavior at microsecond timescales. In *ACM CoNEXT*, pages 133–138. ACM, 2013.

[34] John F. Kim. Resilient roce relaxes rdma requirements. http://kr.mellanox.com/blog/2016/07/resilient-roce-relaxes-rdma-requirements/, 2016.

[35] Radhika Mittal. TIMELY algorithm to compute new rate, 2015. http://radhikam.web.illinois.edu/timely-code-snippet.cc.

[36] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM*, 2015.

[37] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *ACM SIGCOMM*, 2018.

[38] B Montazeri, Yilong Li, Mohammad Alizadeh, and John K Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *ACM SIGCOMM*, 2018.

[39] Timothy Prickett Morgan. The Tug of War between Infiniband and Ethernet, 2017. https://www.nextplatform.com/2017/10/30/tug-war-infiniband-ethernet/.

[40] Jayaram Mudigonda, Praveen Yalagandula, Jeffrey C Mogul, Bryan Stiekes, and Yanick Pouffary. Netlord: A scalable multi-tenant network architecture for virtualized datacenters. In *Proc. of ACM SIGCOMM*, 2011.

[41] Mohammad Noormohammadpour and Cauligi S Raghavendra. Datacenter traffic control: Understanding techniques and tradeoffs. *IEEE Communications Surveys & Tutorials*, 20(2):1492–1525, 2018.

[42] Jim O'Reilly. Future of the Ethernet drive may hinge on NVMe over Ethernet. http://searchstorage.techtarget.com/feature/Future-of-the-Ethernet.

[43] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. *ACM SIGCOMM Computer Communication Review*, 44(4):307–318, 2015.

[44] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

[45] Alex Shpiner, Eitan Zahavi, Vladimir Zdornov, Tal Anker, and Matty Kadosh. Unlocking credit loop deadlocks. In *ACM HotNets*, 2016.

[46] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical dcb for improved data center networks. In *IEEE INFOCOM*, 2014.

[47] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C Snoeren. Practical tdma for datacenter ethernet. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 225–238. ACM, 2012.

[48] Yibo Zhu. NS-3 simulator for RDMA over Converged Ethernet v2 (RoCEv2), 2016. https://github.com/bobzhuyb/ns3-rdma.

[49] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *ACM SIGCOMM*, 2015.

[50] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 479–491. ACM, 2015.

A Theoretical Analysis

We build a fluid model of PCN to exhibit PCN's good performance of fast convergence, fairness, and stability. The main symbols are summarized in Table 2.

A.1 Fluid Model

Suppose N sources share the bottleneck link with capacity C. For each source i $(i = 1, \cdots, N)$ and each CNP generating time $t_k = kT (k = 1, 2, \cdots)$, $R_i(k), \widehat{R_i}(k)$ and $Q(k)$ denote the sending rate and receiving rate of source i, and the queue length in bottleneck link, respectively. Clearly, the queue length $Q(k)$ evolutes as follows

$$Q(k+1) = \max\{0, Q(k) + [\sum R_i(k) - C]T\} \tag{3}$$

As the associated egress port is not paused by its downstream device and excessive packets are accumulated in buffer, we regard the flow through this port is congested. Define the congestion indicator function $p(k)$

$$p(k) = \begin{cases} 0, & if\ Q(k+1) = 0 \\ 1, & if\ Q(k+1) > 0 \end{cases} \tag{4}$$

When N sources share the bottleneck capacity C by the sending ratio $\eta_i(k) = \frac{R_i(k)}{\sum_{j=1}^{N} R_j(k)}$. If the link is underflow, all incoming traffic can arrive their receiver side. Consequently, the receiving rate of each source satisfies

$$\widehat{R_i}(k) = p(k)\eta_i(k)C + (1 - p(k))R_i(k) \tag{5}$$

With probability $p(k)$ and receiving rate $\widehat{R_i}(k)$, source i will change its sending rate and the weight factor according to the corresponding adjustment rule. Thus, we have,

$$\begin{aligned} R_i(k+1) = p(k)\widehat{R_i}(k)(1 - w_{\min}) + \\ (1 - p(k))[R_i(k)(1 - w(k)) + Cw(k)] \end{aligned} \tag{6}$$

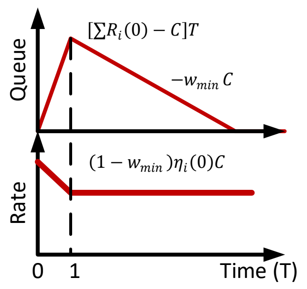
and

$$\begin{aligned} w(k+1) = p(k)w_{\min} + \\ (1 - p(k))[w(k)(1 - w(k)) + w_{\max}w(k)] \end{aligned} \tag{7}$$

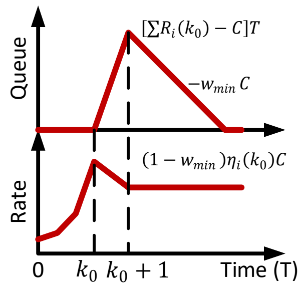
The dynamic behavior of PCN congestion management system can be described using Equation (3), (4), (5), (6) and (7). Based on this fluid model, we analyze PCN's properties in terms of convergence, fairness, and stability.

Variable	Description
R_i	Sending rate of Flow i
η_i	Bandwidth allocation ratio, $\eta_i = \frac{R_i}{\sum R_i}$
$\widehat{R_i}$	Receiving rate of Flow i
w	Weight factor
Q	Bottleneck queue length
T	CNP generating period
k	Sequence of CNP generating periods
C	Bottleneck link capacity

Table 2: Variables of fluid model



(a) $\sum R_i(0) > C$



(b) $\sum R_i(0) \le C$

Figure 15: Convergence of PCN.

A.2 Performance Analysis

A.2.1 Convergence

Without loss of generality, assume the queue associated with bottleneck link is empty and the sending rate of N flows is arbitrary at initial time 0. Hence, there are two cases.

(1) $\sum R_i(0) > C$: If the total rate exceeds the bottleneck capacity, the corresponding queue increases and all flows conduct the rate-decrease adjustment, thus,

$$\begin{cases} Q(1) &= [\sum R_i(0) - C]T \\ R_i(1) &= (1-w_{\min})\eta_i(0)C \end{cases}$$

Note that the total rate $\sum R_i(1) = (1-w_{\min})C < C$, thus the buffer will be drained out in next several periods. When the port is in congestion, the rate-decrease algorithm takes effect. Since $\frac{R_i(1)}{\sum R_i(1)} = \frac{R_i(0)}{\sum R_i(0)}$, we have

$$\begin{cases} Q(k) &= [\sum R_i(0) - C]T - (k-1)w_{\min}CT \\ R_i(k) &= (1-w_{\min})\eta_i(0)C \end{cases} \tag{8}$$

Equation (8) implies that the total sending rate $\sum R_i(k)$ converges to $(1-w_{\min})C$ in one control loop, while the queue length approaches to zero after $\lceil 1 + \frac{\sum R_i(0) - C}{w_{\min}C} \rceil$ control loops. The detail evolutions are illustrated in Fig.15(a).

(2) $\sum R_i(0) \le C$: If the total rate is less than the bottleneck capacity, all flows run the rate-increase algorithm. Eventually, the total sending rate will exceed the link capacity after K_0 control loops, where $K_0 < 10$ according to Fig.10. Therefore, we have $\sum R_i(k_0) > C$ and $Q(k_0) = 0$. Subsequently, the dynamic behavior of both queue and aggregate rate drawn in Fig.15(b) are similar with those in above case.

In a word, PCN can achieve convergence of total rate towards the bottleneck capacity as fast as in only one control loop, and drain out backlog packets.

A.2.2 Fairness

Suppose the above convergence phase ends at the start of k_1 control loop, where the buffer has been drained out, and the sending rate of flow i is increased from $(1-w_{\min})\eta_i(k_0)C$, then,

$$\begin{cases} Q(k_1) &= 0 \\ R_i(k_1) &= (1-w_{\min})^2\eta_i(k_0)C + w_{\min}C \end{cases} \tag{9}$$

and

$$\eta_i(k_1) = \frac{R_i(k_1)}{\sum R_i(k_1)} = \frac{(1-w_{\min})^2\eta_i(k_0) + w_{\min}}{(1-w_{\min})^2 + Nw_{\min}} \tag{10}$$

Note that $\sum R_i(k_1) = [1 + (N-2+w_{\min})w_{\min}]C > C$, the bottleneck link becomes real congested and the RP conducts the rate-decrease adjustment in the next period, thus we have

$$\begin{cases} Q(k_1+1) &= (N-2+w_{\min})w_{\min}CT \\ R_i(k_1+1) &= (1-w_{min})\eta_i(k_1)C \end{cases} \tag{11}$$

Since the aggregate sending rate will become below C, the backlog packets in queue will be drained out at a rate of $w_{\min}C$ per period and the sending rate is kept, then

$$\begin{cases} Q(k_1+k) &= (N-k-1+w_{\min})w_{\min}CT \\ R_i(k_1+k) &= (1-w_{min})\eta_i(k_1)C \end{cases}$$

Obviously, the buffer will become empty again at the starting of k_1+N control loop, and the sending rate of flow i is increased from $(1-w_{\min})\eta_i(k_1)C$, thus

$$\begin{cases} Q(k_1+N) &= 0 \\ R_i(k_1+N) &= (1-w_{\min})^2\eta_i(k_1)C + w_{\min}C \end{cases} \tag{12}$$

and

$$\eta_i(k_1+N) = \frac{(1-w_{\min})^2\eta_i(k_1) + w_{\min}}{(1-w_{\min})^2 + Nw_{\min}} \tag{13}$$

Comparing equation (9) and (12), we can find that PCN repeats the one period of rate-increase and N-1 periods of rate-decrease, as illustrated in Fig.16. And from equation (10) and (12), we also obtain the following dynamic evolution of bandwidth allocation ratio of each flow,

$$\begin{aligned} \eta_i(k_1+kN) &= a\eta_i(k_1+(k-1)N) + b \\ &= a^k\eta_i(k_1) + \textstyle\sum_{j=0}^{k-1} a^j b \\ &= a^{k+1}\eta_i(k_0) + \textstyle\sum_{j=0}^{k} a^j b \end{aligned}$$

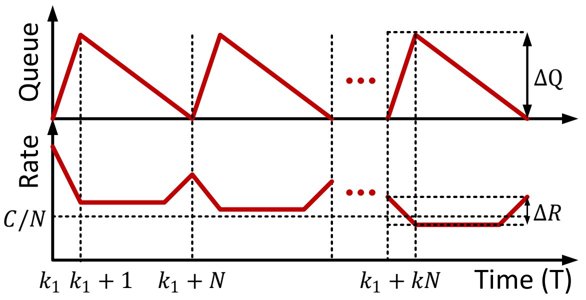


Figure 16: Dynamic behavior of PCN.

where $a = \frac{(1-w_{\min})^2}{(1-w_{\min})^2+Nw_{\min}} \in (0,1)$, $b = \frac{w_{\min}}{(1-w_{\min})^2+Nw_{\min}}$. Consequently, as $k \to \infty$, there is

$$\eta_i(k_1+Nk) \to \frac{b}{1-a} = \frac{1}{N} \tag{14}$$

That is, PCN can always achieve fair bandwidth allocation regardless of the initial sending rates of flows and parameter settings.

A.2.3 Stability

Finally, we show the steady state behavior of PCN. As illustrated in Fig.16, the queue length varies between 0 and $Q(k_1+1)$ periodically. Based on equation (11), the maximal queue oscillation ΔQ satisfies

$$\Delta Q = Q(k_1+1) = (N-2+w_{\min})w_{\min}CT \tag{15}$$

Similarly, the sending rate also changes in each N control loops. As Fig.16 shows, the aggregate rate increases in k_1+kN period and decreases in k_1+kN+1 period. Note that each flow achieves fair bandwidth allocation ratio in the steady state, i.e., $\eta_i \to \frac{1}{N}$, thus we can obtain the following derivation based on equation (11) and (12),

$$\begin{aligned} R_i(k_1+kN) &= (1-w_{\min})\eta_i C + w_{\min}C \\ &\to [1+(N-1)w_{\min}]\tfrac{C}{N} \\ R_i(k_1+kN+1) &= (1-w_{\min})\eta_i C \\ &\to (1-w_{\min})\tfrac{C}{N} \end{aligned}$$

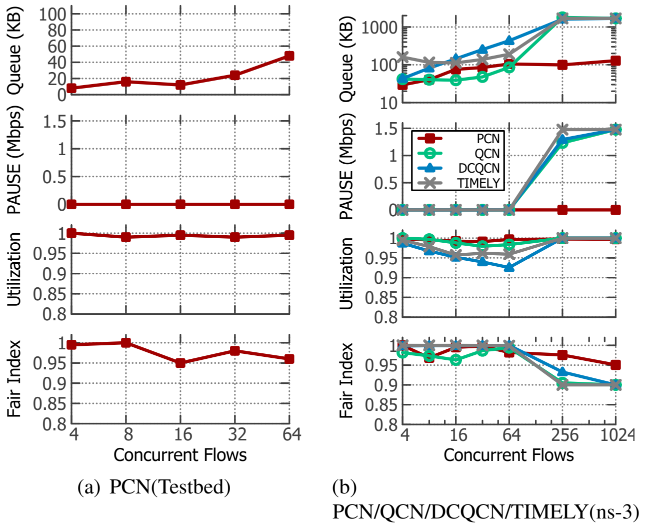
Therefore, the rate oscillation ΔR_i around the fair share $\frac{C}{N}$ satisfies

$$\Delta R_i \to w_{\min}C \tag{16}$$

Equation (15) and (16) indicate the oscillations of both the queue and rate are bounded in steady state, i.e., the stability of PCN is fine.

B External Evaluations

In this section, we will explore how PCN performs in artificial cases, including *flow scalability*, *adversarial traffic*, *multiple bottlenecks*, *multiple priorities* and *deadlock*.



(a) PCN(Testbed) (b) PCN/QCN/DCQCN/TIMELY(ns-3)

Figure 17: Flow scalability test.

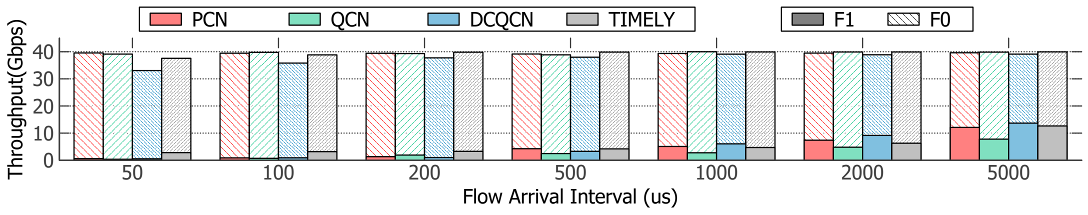
B.1 Flow Scalability

Using the simple 3-dumbbell topology in §7.1, we vary the number of flows from 4 to 64 (testbed) and 1024 (ns-3) to test the performance of PCN under more flows. The queue length, pause rate, link utilization and fairness are measured and calculated, and the results are presented in Fig.17.

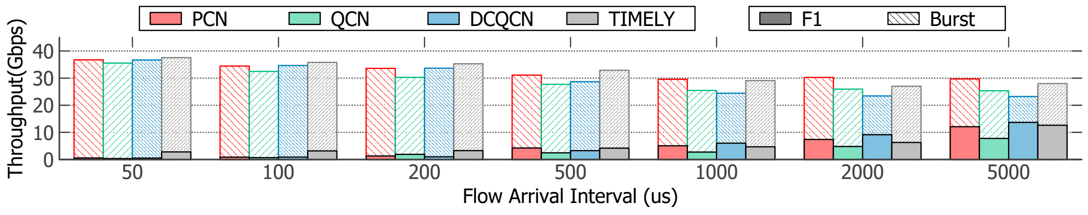
First, we measure the average queue occupancy and pause rate as the number of concurrent flows increases. In PCN, the average queue length is no more than 60KB and 100KB in the testbed and ns-3 simulator, respectively. At the same time, there are no PAUSE frames generated. In contrast, with $4 \sim 256$ flows, DCQCN's queue length grows with the number of flows, QCN and TIMELY keep the queue length around $50KB \sim 100KB$ and $100 \sim 200KB$, respectively. But QCN, DCQCN and TIMELY maintain a very high queue occupancy beyond 256 flows, which indicates the end-to-end congestion control fail to take effect. As for QCN, DCQCN and TIMELY, PFC is rarely triggered when the number of flows is less than 256, but persistent PAUSE frames are generated.

Second, we measure the utilization of bottleneck link. PCN achieves near 100% utilization in all case with both testbed and ns-3 simulator. QCN, DCQCN and TIMELY have a little under-utilization with the increase of concurrent flows, but recover full utilization with more than 256 flows. However, this recovery of link utilization is due to PFC rather than the end-to-end congestion control schemes.

Finally, we calculate the Jain's fairness index [30] using the throughput of each flow at $500ms$ interval. With a large number of flows, the fairness index of QCN, DCQCN and TIMELY drops significantly. Because they can not prevent PFC from persistent triggers, the inherent unfairness problem of PFC exhibits. On the opposite, PCN achieves good fairness in all cases.

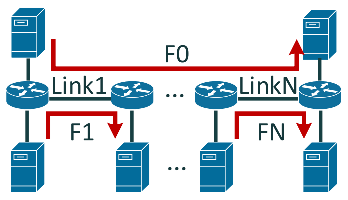


(a) Link S0→ S1

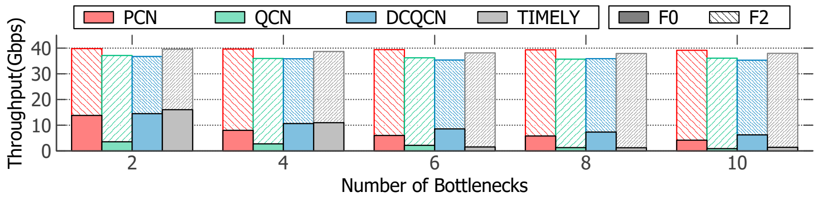


(b) Link S1→R1

Figure 18: Performance of various schemes under adversarial traffic.



(a) Parking-lot Topology



(b) Throughput of Link1

Figure 19: Performance of various schemes under multi-bottleneck scenario.

B.2 Adversarial Traffic

Subsequently, we test PCN using an adversarial traffic pattern. In the basic scenario in Fig.1, we set long flows F0 and F1 transmit persistently, and burst flows from H2∼H15 to R1 enter and exit the network at different intervals, varying from 50μs (1 control loop) to 5000μs (100 control loops). We simulate PCN, QCN, DCQCN and TIMELY, and measure the throughput of the two bottlenecks (link S0→S1 and S1→R1) and different flows. The results are drawn in Fig.18.

The first bottleneck, link S0→S1, is irrelative with the burst flows. Fig.18(a) shows that under PCN, QCN and TIMELY, link S0→S1 can achieve near-full utilization. But when the burst flows becomes more frequent, DCQCN trends to loss as high as 15% of throughput. This is because switch S1 pauses S0 when the burst flows make P2|S1 congested, and DCQCN conducts improper rate decrease for the victim flow F0.

The second bottleneck, link S1→R1, is frequently interrupted by the burst flows. Fig.18(b) exhibits that when the flow arrival interval shrinks, the congestion-relative flow F1 occupies lower throughput but the link utilization becomes larger. The performance issue occurs when the flow arrival interval is a little large (>500μs, 10 control loops). This means, their rate increase phase is interrupted by new-arrival flows. We can see that PCN keeps the link throughput at 30Gbps, while QCN, DCQCN and TIMELY remains at 23Gbps, 25Gbps and 29Gbps, respectively. That is, PCN can alleviate, but not eliminate, the interruption from adversarial traffic.

B.3 Multiple Bottlenecks

In a multi-bottleneck scenario, the NP-ECN method of PCN's CP may encounter several issues. On one hand, when the first congestion point marks ECN on all packets, the second congestion point may be paused, thus some flows are the victim but they have been marked with ECN already. On the other hand, flows through multiple congestion points may have a larger probability to be marked with ECN, resulting in unfairness. To test how PCN performs in multiple bottleneck scenario, we conduct a series of simulations using the parking lot topology in Fig.19(a). There are N bottlenecks and $N+1$ flows, where we set $N = 2,4,6,8,10$. F0 passes all the bottlenecks while other flows pass only one bottleneck. We measure the throughput of F0 and F1, and their sum is the throughput of link1. The result is drawn in Fig.19(b). Obviously, link1 achieves the similar utilization regardless of the number of bottlenecks. PCN can always provide more than 98% of link utilization, while the link utilization under QCN and DCQCN

Scheme	S(P1)	M(P2)	L(P3)	XL(P4)
PCN	0	0	0.10	171.49
QCN	0	0	0.28	110.49
DCQCN	0	0	0.26	175.02
TIMELY	0	0	1.09	210.36

Table 3: Generating rate of PFC PAUSEs for multiple priorities.

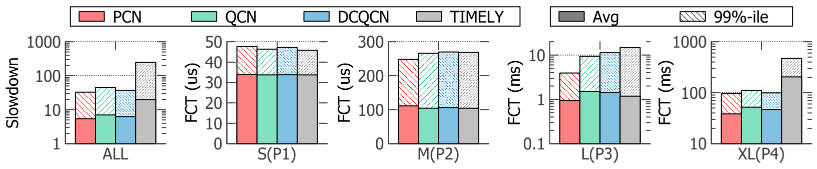


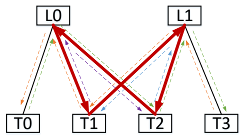
Figure 20: Average/99%-ile flow completion time for multiple priorities.

and TIMELY is 90%, 88% and 95%, respectively. Meanwhile, F0 is allocated less bandwidth than F2. Actually, the bandwidth allocation of PCN conforms to *proportional fairness*, where F0 obtains about $\frac{1}{N+1}$ of the capacity. QCN allocates F0 less than the proportional fairness. DCQCN allocates F0 more than the proportional fairness, but also less than the *max-min fairness*, i.e, half of the capacity.

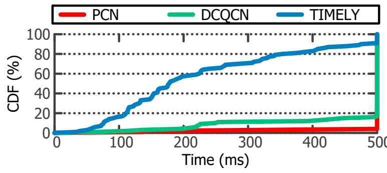
B.4 Multiple Priorities

The switching fabric in data center typically provides multiple priorities to improve performance, especially for minimizing flow completion time. The principle "short flow first" has been adopted in a series of works such as pHost [24], pFabric [14] and PIAS [17]. However, the concurrent burst in higher priority may trigger more PAUSE in lower priority, and impact the end-to-end congestion control schemes. To demonstrate and confirm this fact, we configure W1 and repeat the simulation in § 7.4, where the flows are classified into four priorities according to their size, namely, the S size flows are in the first priority and the XL flows are gathered in the fourth priority. The switches forward packets following the strict priority scheduling.

The generating rate of PFC PAUSEs and FCT for different priorities are listed in Table 3 and shown in Fig.20. For S and M flows in these two high priorities, few PFC PAUSE messages generate regardless of congestion control schemes. Thus, these flows can obtain almost the same FCT under three congestion control schemes. On the contrary, for the L and XL flows in the two low priorities, PFC PAUSEs can not be avoided. In this case, PCN triggers less PAUSE compared with DCQCN and TIMELY. QCN reduces PAUSE generated for XL flows by underutilizing available bandwidth. PCN outperforms the other three schemes in speeding up the overall flow completion time.



(a) Deadlock Topology



(b) Statistical Results

Figure 21: Performance of various schemes under deadlock scenario.

B.5 Deadlock Scenario

A common concern in Lossless Ethernet is that PAUSE can lead to deadlocks [28]. To explore PCN's effort to avoiding deadlock, we conduct a simple simulation using the topology illustrated in Fig.21(a). It comes from one pod in the clos network used in § 7.4, but link L0-T3 and link L1-T0 are failed, such that there is a *cycle buffer dependency* (CBD) as the red line draws. We simulate PCN, DCQCN and TIMELY with the W2 workload. The target load is 0.6 at ToR downlinks with in-cast ratio ranging from 1 to 15. Each scheme is tested for 1000 times and every simulation lasts for 500ms. We record the time when deadlock occurs, and draw the statistical results in Fig.21(b). Among the 1000 simulations, PCN only encounters with deadlock for 28 times, while DCQCN and TIMELY are deadlocked for 134 and 870 times, respectively. The advantage of PCN comes from the positive effect of mitigating PFC triggers and stopping congestion spreading.

Measuring Congestion in High-Performance Datacenter Interconnects

Saurabh Jha[1], Archit Patke[1], Jim Brandt[2], Ann Gentile[2], *Benjamin Lim*[1],
Mike Showerman[3], Greg Bauer[3], Larry Kaplan[4], Zbigniew Kalbarczyk[1], William Kramer[1,3], Ravi Iyer[1]

[1]*University of Illinois at Urbana-Champaign,* [2]*Sandia National Lab,*
[3]*National Center for Supercomputing Applications,* [4]*Cray Inc.*

Abstract

While it is widely acknowledged that network congestion in High Performance Computing (HPC) systems can significantly degrade application performance, there has been little to no quantification of congestion on credit-based interconnect networks. We present a methodology for detecting, extracting, and characterizing regions of congestion in networks. We have implemented the methodology in a deployable tool, *Monet*, which can provide such analysis and feedback at runtime. Using *Monet*, we characterize and diagnose congestion in the world's largest 3D torus network of *Blue Waters*, a 13.3-petaflop supercomputer at the National Center for Supercomputing Applications. Our study deepens the understanding of production congestion at a scale that has never been evaluated before.

1 Introduction

High-speed interconnect networks (HSN), e.g., Infiniband [48] and Cray Aries [42]), which uses credit-based flow control algorithms [32, 61], are increasingly being used in high-performance datacenters (HPC [11] and clouds [5, 6, 8, 80]) to support the low-latency communication primitives required by extreme-scale applications (e.g., scientific and deep-learning applications). Despite the network support for low-latency communication primitives and advanced congestion mitigation and protection mechanisms, significant performance variation has been observed in production systems running real-world workloads. While it is widely acknowledged that network congestion can significantly degrade application performance [24, 26, 45, 71, 81], there has been little to no quantification of congestion on such interconnect networks to understand, diagnose and mitigate congestion problems at the application or system-level. In particular, tools and techniques to perform runtime measurement and characterization and provide runtime feedback to system software (e.g., schedulers) or users (e.g., application developers or system managers) are generally not available on production systems. This would require continuous system-wide, data collection on the state of network performance and associated complex analysis which may be difficult to perform at runtime.

The core contributions of this paper are (a) a methodology, including algorithms, for quantitative characterization of congestion of high-speed interconnect networks; (b) introduction of a deployable toolset, *Monet* [7], that employs our congestion characterization methodology; and (c) use of the the methodology for characterization of congestion using 5 months of operational data from a 3D torus-based interconnect network of *Blue Waters* [1, 27, 60], a 13.3-petaflop Cray supercomputer at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. The novelty of our approach is its ability to use *percent time stalled* (P_{Ts})[1] metric to detect and quantitatively characterize congestion hotspots, also referred to as *congestion regions (CRs)*, which are group of links with similar levels of congestion.

The *Monet* tool has been experimentally used on NCSA's *Blue Waters*. *Blue Waters* uses a Cray Gemini [21] 3D torus interconnect, the largest known 3D torus in existence, that connects 27,648 compute nodes, henceforth referred to as *nodes*. The proposed tool is not specific to Cray Gemini and *Blue Waters*; it can be deployed on other k-dimensional mesh or toroidal networks, such as TPU clouds [3], Fujitsu TOFU network-based [18, 20] K supercomputer [70] and upcoming post-K supercomputer [10][2]. The key components of our methodology and the *Monet* toolset are as follows:

Data collection tools: On *Blue Waters*, we use vendor-provided tools (e.g., `gpcdr` [35]), along with the Lightweight Distributed Metric Service (LDMS) monitoring framework [17]. Together these tools collect data on (a) the network (e.g., transferred/received bytes, congestion metrics, and link failure events); (b) the file system traffic (e.g., read/write bytes); and (c) the applications (e.g., start/end time). We are released raw network data obtained from *Blue Waters* [57] as well as the associated code for generating *CRs* as artifacts with this paper [7]. To the best of our knowledge, this is the first

[1]P_{Ts}, defined formally in Section 2, approximately represents the intensity of congestion on a link, quantified between 0% and 100%.

[2]The first post-K supercomputer is scheduled to be deployed in 2021.

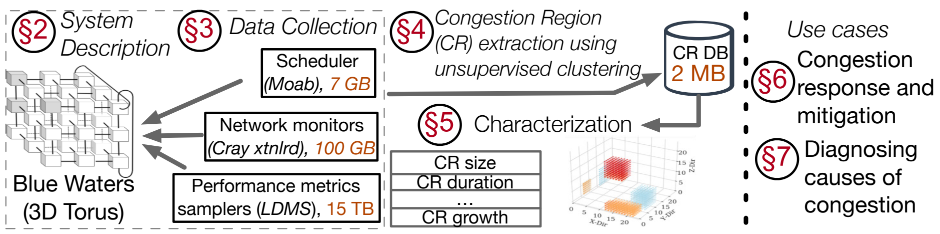

§2 System Description
§3 Data Collection
§4 Congestion Region (CR) extraction using unsupervised clustering
CR DB 2 MB
Use cases
§6 Congestion response and mitigation
§7 Diagnosing causes of congestion
Scheduler (Moab), 7 GB
Network monitors (Cray xtnlrd), 100 GB
Performance metrics samplers (LDMS), 15 TB
Blue Waters (3D Torus)
§5 Characterization
CR size
CR duration
...
CR growth

Figure 1: Characterization and diagnosis workflow for interconnection-networks.

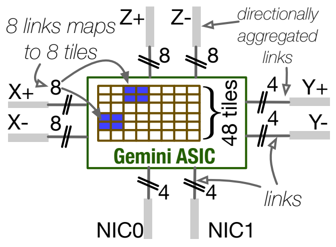


Figure 2: Cray Gemini 48-port switch.

such large-scale network data release for an HPC high-speed interconnect network that uses credit-based flow control.

A network hotspot extraction and characterization tool, which extracts *CRs* at runtime; it does so by using an unsupervised region-growth clustering algorithm. The clustering method requires specification of congestion metrics (e.g., percent time stalled (P_{Ts}) or stall-to-flit ratios) and a network topology graph to extract regions of congestion that can be used for runtime or long-term network congestion characterization.

A diagnosis tool, which determines the cause of congestion (e.g., link failures or excessive file system traffic from applications) by combining system and application execution information with the CR characterizations. This tool leverages outlier-detection algorithms combined with domain-driven knowledge to flag anomalies in the data that can be correlated with the occurrence of CRs.

To produce the findings discussed in this paper, we used 5 months of operational data on *Blue Waters* representing more than 815,006 unique application runs that injected more than 70 PB of data into the network. Our key findings are as follows:

- *While it is rare for the system to be globally congested, there is a continuous presence of highly congested regions (CRs) in the network, and they are severe enough to affect application performance*. Measurements show that (a) for more than 56% of system uptime, there exists at least one highly congested CR (i.e., a CR with a $P_{Ts} > 25\%$), and that these CRs have a median size of 32 links and a maximum size of 2,324 links (5.6% of total links); and (b) highly congested regions may persist for more than 23 hours, with a median duration time of 9 hours[3]. With respect to impact on applications, we observed 1000-node production runs of the NAMD [77] application [4] slowing down by as much as 1.89× in the presence of high congestion compared to median runtime of 282 minutes.
- *Once congestion occurs in the network, it is likely to persist rather than decrease, leading to long-lived congestion in the network*. Measurements show that once the network has entered a state of high congestion ($P_{Ts} > 25\%$), it will persist in high congestion state with a probability of 0.87 in the next measurement window.
- *Quick propagation of congestion can be caused by network component failures.* Network component failures (e.g., network router failures) that occur in the vicinity of a large-scale application can lead to high network congestion within minutes of the failure event. Measurements show that 88% of directional link failures [5] caused the formation of CRs with an average $P_{Ts} \geq 15\%$.
- *Default congestion mitigation mechanisms have limited efficacy.* Our measurements show that (a) 29.8% of the 261 triggers of vendor-provided congestion mitigation mechanisms failed to alleviate long-lasting congestion (i.e., congestion driven by continuous oversubscription, as opposed to isolated traffic bursts), as they did not address the root causes of congestion; and (b) vendor-provided mitigation mechanisms were triggered in 8% (261) of the 3,390 high-congestion events identified by our framework. Of these 3,390 events, 25% lasted for more than 30 minutes. This analysis suggests that augmentation of the vendor-supplied solution could be an effective way to improve overall congestion management.

In this paper, we highlight the utility of congestion regions in the following ways:

- We showcase the effectiveness of *CRs* in detecting long-lived congestion. Based on this characterization, we propose that *CR* detection could be used to trigger congestion mitigation responses that could augment the current vendor-provided mechanisms.
- We illustrate how *CRs*, in conjunction with network traffic assessment, enable congestion diagnosis. Our diagnosis tool attributes congestion cause to one of the following: (a) system issues (such as launch/exit of application), (b) failure issues (such as network link failures), and (c) intra-application issues (such as changes in communication patterns within an application). Such a diagnosis allows system managers to take cause-specific mitigating actions.

This paper's organization is illustrated in Figure 1. We present background information on the Gemini network, performance data, and congestion mitigation mechanisms in Section 2. In Section 3, we present our data collection methodology and tools. In Section 4, we present our methodology for characterizing congestion. We present our measurement-

[3]Note that *Blue Waters* allows applications to run for a maximum of 48 hours.
[4]NAMD is the top application running on Blue Waters consuming 18% of total node-hours [58].

[5]see Section 5.4 for the definition of directional link.

driven congestion characterization results in Section 5. In Section 6, we discuss the further utility of our methodology to inform targeted responses, and in Section 7, we discuss its use in diagnosing the root causes of congestion. We address related work in Section 8 and conclude in Section 9.

2 Cray Gemini Network and Blue Waters

A variety of network technologies and topologies have been utilized in HPC systems (e.g., [19, 21, 31, 36, 42, 59, 62, 75]). Depending on the technology, routing within these networks may be statically defined for the duration of a system boot cycle, or may dynamically change because of congestion and/or failure conditions. More details on HPC interconnects can be found in Appendix A. The focus of this paper is on NCSA's Cray XE/XK *Blue Waters* [1] system, which is composed of 27,648 nodes and has a large-scale (13,824 x 48 port switches) Gemini [21] 3D torus (dimension 24x24x24) interconnect. It is a good platform for development and validation of congestion analysis/ characterization methods as:

- It uses directional-order routing, which is predominantly static[6]. From a traffic and congestion characterization perspective, statically routed environments are easier to validate than dynamic and adaptive networks.
- *Blue Waters* is the best case torus to study since it uses topology-aware scheduling (TAS) [41, 82], discussed later in this section, which has eliminated many congestion issues compared to random scheduling.
- *Blue Waters* performs continuous system-wide collection and storage of network performance counters.

2.1 Gemini Network

In Cray XE/XK systems, four *nodes* are packaged on a *blade*. Each *blade* is equipped with a mezzanine card. This card contains a pair of *Gemini [21] ASICs*, which serve as network switches. The Gemini switch design is shown in Figure 2. Each Gemini ASIC consists of 48 *tiles*, each of which provide a duplex link. The switches are connected with one another in 6 directions, X+/-, Y+/- and Z+/-, via multiple links that form a 3D torus. The number of links in a direction, depends on the direction as shown in the figure; there are 8 each in X+/- and, Z+/- and 4 each in Y+/-. It is convenient to consider all links in a given direction as a *directionally aggregated link*, which we will henceforth call a *link*. The available bandwidth on a particular link is dependent on the link type, i.e., whether the link connects compute cabinets or *blades*, in addition to the number of tiles in the link [76]. X, Y links have aggregate bandwidths of 9.4 GB/s and 4.7 GB/s, respectively, whereas Z links are predominantly 15 GB/s, with 1/8 of them at 9.4 GB/s. Traffic routing in the Gemini network is largely static and changes only when failures occur that need to be routed around. Traffic is directionally routed in the X, Y, and Z dimensions, with the shortest path in terms of hops in + or - chosen for each direction. A deterministic rule handles tie-breaking.

To avoid data loss in the network [7], the Gemini HSN uses a credit-based flow control mechanism [61], and routing is done on a per-packet basis. In credit-based flow control networks, a source is allowed to send a quantum of data, e.g., a flit, to a next hop destination only if it has a sufficient number of credits. If the source does not have sufficient credits, it must stall (wait) until enough credits are available. Stalls can occur in two different places: within the switch (resulting in a *inq stall*) or between switches (resulting in an *credit stall*).

Definition 1 ***: A Credit stall*** *is the wait time associated with sending of a flit from an output buffer of one switch to an input buffer of another across a link.*

Definition 2 ***: An Inq stall*** *is the wait time associated with sending of a flit from the output buffer of one switch port to an input buffer of another between tiles within the same network switch ASIC.*

Congestion in a Gemini-based network can be characterized using both *credit* and *inq* stall metrics. Specifically, we consider the *Percent Time Stalled* as a metric for quantifying congestion, which we generically refer to as the *stall value*.

Definition 3 ***: Percent Time Stalled*** *(P_{Ts}) is the average time spent stalled (T_{is}) over all* tiles *of a directional network link or individual intra-Gemini switch link over the same time interval (T_i): $P_{Ts} = 100 * T_{is}/T_i$.*

Depending on the network topology and routing rules, (a) an application's traffic can pass through switches not directly associated with its allocated nodes, and multiple applications can be in competition for bandwidth on the same network links; (b) stalls on a link can lead to back pressure on prior switches in communication routes, causing congestion to spread; and (c) the initial manifestation location of congestion cannot be directly associated with the cause of congestion. Differences in available bandwidth along directions, combined with the directional-order routing, can also cause back pressure, leading to varying levels of congestion along the three directions.

2.2 Congestion Mitigation

Run-time evaluations that identify localized areas of congestion and assess congestion duration can be used to trigger *Congestion Effect Mitigating Responses (CEMRs)*, such as resource scheduling, placement decisions, and dynamic application reconfiguration. While we have defined a CEMR as a response that can be used to minimize the negative effects of network congestion, Cray provides a software mechanism [33] to directly alleviate the congestion itself. When a

[6]When network-link failures occur, network routes are recomputed; that changes the route while the system is up.

[7]The probability of loss of a quantum of data in credit-flow networks is negligible and mostly occurs due to network-related failures.

variety of network components (e.g., tiles, NICs) exceeds a high-watermark threshold with respect to the ratio of stalls to forwarded flits, the software instigates a *Congestion Protection Event* (CPE), which is a *throttling* of injection of traffic from all NICs. The CPE mechanism limits the aggregate traffic injection bandwidth over *all* compute nodes to less than what can be ejected to a *single* node. While this ensures that the congestion is at least temporarily alleviated, the network as a whole is drastically under-subscribed for the duration of the *throttling*. As a result, the performance of all applications running on the system can be significantly impacted. *Throttling* remains active until associated monitored values and ratios drop below their low-watermark thresholds. Applications with sustained high traffic injection rates may induce many CPEs, leading to significant time spent in globally throttling. Bursts of high traffic injection rates may thus trigger CPEs, due to localized congestion, that could have been alleviated without the global negative impact of *throttling*. There is an option to enable the software to terminate the application that it determines is the top congestion candidate, though this feature is not enabled on the *Blue Waters* system. The option to terminate application in a production environment is not acceptable to most developers and system managers as it will lead to loss of computational node-hours used by the application after the last checkpoint.

While some of this congestion may be alleviated by CEMRs such as feedback of congestion information to applications to trigger rebalancing [29] or to scheduling/resource managers to preferentially allocate nodes (e.g., via mechanisms such as slurm's [79] node weight), some may be unavoidable since all networks have finite bandwidth.

On *Blue Waters* a topology-aware scheduling (TAS) [41, 82] scheme is used to decrease the possibility of application communication interference by assigning, by default [12], node allocations that are constrained within small-convex prisms with respect to the HSN topology. Jobs that exceed half a torus will still route outside the allocation and possibly interfere with other jobs and vice versa; a non-default option can be used to avoid placement next to such jobs. The I/O routers represent fixed, and roughly evenly distributed, proportional portions of the storage subsystem. Since the storage subsystem components, including I/O routers, are allocated (for writes) in a round robin (by request order) manner independent of TAS allocations, storage I/O communications will generally use network links both within and outside the geometry of the application's allocation and can also be a cause of interference between applications.

3 Data Sources and Data Collection Tools

This section describes the datasets and tools used to collect data at scale to enable both runtime and long-term characterization of network congestion. We leverage vendor-provided and specialized tools to enable collection and real-time streaming of data to a remote compute node for analysis and characterization. Data provided or exposed on all Cray Gemini systems includes: OS and network performance counter data, network resilience-related logs, and workload placement and status logs. In this study, we used five months (Jan 01 to May 31, 2017) of production network performance-related data (15 TB), network resilience-related logs (100 GB), and application placement logs (7 GB). Note that the methodologies addressed in this work rely only on the availability of the data, independent of the specific tools used to collect the data.

Network Performance Counters: Network performance-related information on links is exposed via Cray's `gpcdr` [35] kernel module. Lustre file system and RDMA traffic information is exposed on the nodes via `/proc/fs` and `/proc/kgnilnd`. It is neither collected nor made available for analysis via vendor-provided collection mechanisms. On Blue Waters, these data are collected and transported off the system for storage and analysis via the Lightweight Distributed Metric Service (LDMS) monitoring framework [17]. In this work, we use the following information: directionally aggregated network traffic (bytes and packets) and length of stalls due to credit depletion; Lustre file system read and write bytes; and RDMA bytes transmitted and received. LDMS samplers collect those data at 60-second intervals and calculate derived metrics, such as the percent of time spent in stalls (P_{Ts}) and percent of total bandwidth used over the last interval. LDMS daemons synchronize their sampling to within a few *ms* (neglecting clock skew) in order to provide coherent *snapshots* of network state across the whole system.

Network Monitoring Logs: Network failures and congestion levels are monitored and mitigated by Cray's *xtnlrd* software. This software further logs certain network events in a well-known format in the *netwatch* log file. Significant example log lines are provided in Cray documents [33, 34]. Regular expression matching for these lines is implemented in LogDiver [66], a log-processing tool, which we use to extract the occurrences, times, and locations of link failures and CPEs.

Workload Data: Blue Waters utilizes the Moab scheduler, from which application queue time, start time, end time, exit status, and allocation of nodes can be obtained. The workload dataset contains information about 815,006 application runs that were executed during our study period. A detailed characterization of *Blue Waters* workloads can be found in Appendix B and *Blue Waters* workload study [58].

Note that we will only be releasing network data. Worload data and network monitoring logs will not be released due to privacy and other concerns.

4 *CR* Extraction and Characterization Tool

This section first describes our motivation for choosing congestion regions (CRs) as a driver for characterizing network congestion, and then describes our methodology (implemented as the *Monet* tool) for extracting *CRs* over each data collection interval and the classification of those CRs based on severity.

4.1 Why Congestion Regions?

We seek to motivate our choice to characterize congestion regions (*CRs*) and the need for estimates for severity in terms of the stall values. We first show that the charcterization of hotspot links individually do not reveal the spatial and growth characteristics which is needed for diagnosis. Then, we show how characterizing *CRs* is meaningful.

Characterizing hotspot links individually do not reveal regions of congestion. Figure 3 characterizes the median, 99%ile and 99.9%ile duration of the hotspot links by generating the distribution of the duration for which a link persists to be in congestion at $P_{Ts} \geq P_{Ts}$Threshold value. For example, 99.9%ile duration for hotspot links with $P_{Ts} \geq 30$ is 400 minutes (6.67 hours). The measurements show that the median duration of hotspot link at different P_{Ts} thresholds is constantly at ~ 0, however, 99.9%ile duration of hotspot links linearly decreases with increasing P_{Ts} threshold value. Although such characterizations are useful to understand congestion at link-level, they hide the spatial characteristics of congestion such as the existence of multiple pockets of congestion and their spread and growth over time. The lack of such information makes it difficult to understand congestion characteristics and their root cause.

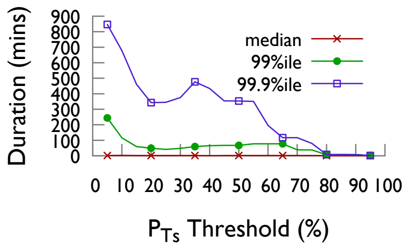
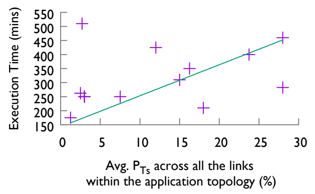
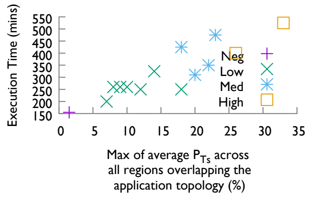


Figure 3: Duration of congestion on links at different P_{Ts} thresholds



(a) Not using **CRs**



(b) Using **CRs**

Figure 4: Correlating congestion with NAMD application runtime

***CRs* captures relationship between congestion-level and application slowdown efficiently.** In order to determine possible severity values and show effectiveness of *CRs* in determining application slowdown, we extracted from the production Blue Waters dataset a set of NAMD [77][8] runs each of which ran on 1000 nodes with the same input parameters. We chose NAMD because it consumes approximately 18% of total node-hours available on Blue Waters[9]. Figure 4a shows the execution time of each individual run with respect to the *average* P_{Ts} over all links within the allocated application topology. (Here we leverage TAS to determine severity value estimates based on the values within the allocation; that is not a condition for the rest of this work.) Figure 4a shows that execution time is perhaps only loosely related to the average P_{Ts}; with correlation of 0.33 . In contrast, 4b shows the relationship of the application execution time with the *maximum* average P_{Ts} over all *CRs* (defined in 4.2) within the allocated topology; with correlation of 0.89. In this case, execution time increases with increasing maximum of average P_{Ts} over all regions. We found this relationship to hold for other scientific applications. This is a motivating factor for the extraction of such *congestion regions* (*CRs*) as indicators of 'hot-spots' in the network. We describe the methodology for *CR* extraction in the next section.

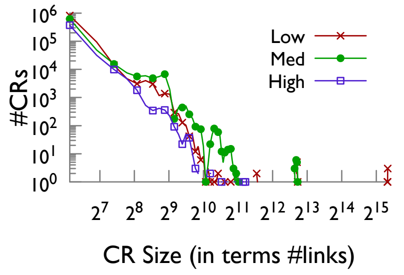
In addition, we selected approximate ranges of P_{Ts} values, corresponding to increasing run times, to use as estimates for the severity levels as these can be easily calculated, understood and compared. These levels are indicated as symbols in the figure. Explicitly, we assign 0-5% average P_{Ts} in a *CR* as *Negligible* or *'Neg'*, 5-15% as *'Low'*, 15-25% as *'Medium'*, and > 25% as *'High'*. These are meant to be qualitative assignments and not to be rigorously associated with a definitive performance variation for all applications in all cases, as the network communication patterns and traffic volumes vary among HPC applications. We will use these ranges in characterizations in the rest of this work. More accurate determinations of impact could be used in place of these in the future, without changing the validity of the *CR* extraction technique.

4.2 Extracting Congestion Regions

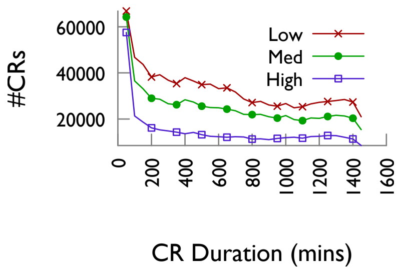
We have developed an unsupervised clustering approach for extracting and localizing regions of congestion in the network by segmenting the network into groups of links with similar congestion values. The clustering approach requires the following parameters: (a) network graph (G), (b) congestion measures (v_s for each vertex v in G), (c) neighborhood distance metric (d_δ), and (d) stall similarity metric (d_λ). The network is represented as a graph G. Each link in the network is represented as a vertex v in G, and two vertices are connected if the corresponding network links are both connected to the same switch (i.e., the switch is an edge in the graphs). For each vertex v, the congestion measures(s) are denoted by the vector v_s, which is composed of credit stalls and inq

[8]NAMD has two different implementations: (a) uGNI shared memory parallel (SMP)-based, and (b) MPI-based. In this work, unstated NAMD refers to uGNI SMP-based implementation. uGNI is user level Generic Network Interface [83].

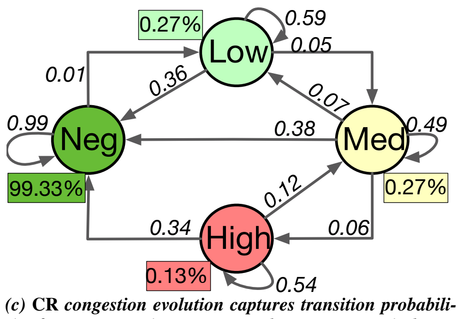
[9]This was best effort extraction and the NAMD application runs may not be exactly executing the same binary or processing the same data, as user may have recompiled the code with a different library or used the same name for dataset while changing the data. There is limited information to extract suitable comparable runs from historical data that are also subject to allocation and performance variation.



(a) Distribution of CR sizes.



(b) Distribution of CRs.



(c) **CR** *congestion evolution captures transition probabilities from one severity* **state** *to another. Percentage in boxes indicates percentage of total link-hours spent in that* **state**.

Figure 5: **CR** *size, duration, evolution characterization. # of* **CRs** *across* **'Low'**, **'Medium'**, *and* **'High'** *are 9.4e05, 7.3e05, and 4.2e05 respectively.*

stalls, which we use independently. Distance metrics d_δ and d_λ are also required, the former for calculating distances between two vertices and the latter for calculating differences among the stalls v_s. We assign each vertex the coordinate halfway between the logical coordinates of the two switches to which that vertex is immediately connected, and we set d_δ to be the L1 norm between the coordinates. Since the Blue Waters data consists of directionally aggregated information as opposed to counters on a per-tile-link (or buffer) basis, then, in our case, d_λ is simply the absolute difference between the two credit-stall or the two inq-stall values of the links, depending on what kinds of regions are being segmented. We consider credit and inq stalls separately to extract *CRs*, as the relationship between the two types of stalls is not immediately apparent from the measurements, and thus require two segmentation passes. Next, we outline the segmentation algorithm.

Segmentation Algorithm The segmentation algorithm has four stages which are executed in order, as follows.

- Nearby links with similar stall values are grouped together. Specifically, they are grouped into the equivalence classes of the reflexive and transitive closure of the relation $\sim_r$ defined by $x \sim_r y \Leftrightarrow d_\delta(x,y) \leq \delta \wedge d_\lambda(x_s - y_s) \leq \theta_p$, where x, y are vertices in G, and δ, θ_p are thresholds for distance between vertices and stall values, respectively.
- Nearby regions with similar average stall values, are grouped together through repetition of the previous step, but with regions in place of individual links. Instead of using the link values v_s, we use the average value of v_s over all links in the region, and instead of using θ_p, we use a separate threshold value θ_r.
- CRs that are below the size threshold σ are merged into the nearest region within the distance threshold δ.
- Remaining CRs with $< \sigma$ links are discarded, so that regions that are too small to be significant are eliminated.

The optimum values for the parameters used in segmentation algorithms, except for δ, were estimated empirically by knee-curve [63] method, based on the number of regions produced. Using that method, the obtained parameter values [10] are: (a) $\theta_p = 4$, (b) $\theta_r = 4$, and (c) $\sigma = 20$. In [63], the authors conclude that the optimum sliding window time is the knee of the curve drawn between the sliding window time and the number of clusters obtained using a clustering algorithm. This decreases truncation errors (in which a cluster is split into multiple clusters because of a small sliding window time) and collision errors (in which two events not related to each other merge into a single cluster because of a large sliding window time). We fixed δ to be 2 in order to consider only links that are two hops away, to capture the local nature of congestion [47]. It should be noted that the region clustering algorithm may discard small isolated regions (size $\leq \sigma$) of high congestion. If such *CRs* do cause high interference, they will grow over time and eventually be captured.

Our algorithm works under several assumptions: (a) congestion spreads locally, and (b) within a CR, the stall values of the links do not vary significantly. These assumptions are reasonable for k-dimensional toroids that use directional-order routing algorithm. The methodology used to derive *CRs* is not dependent on the resource allocation policy (such as TAS). The proposed extraction and its use for characterization is particularly suitable for analysis of network topologies that use directional- or dimensional-order routing. In principle, the algorithm can be applied to other topologies (such as mesh and high-order torus networks) with other metrics (such as stall-to-flit ratio). Furthermore, the region extraction algorithm does not force any shape constraints; thus *CRs* can be of any arbitrary shape requiring us to store each vertex associated with the *CR*. In this work, we have configured the tool to store and display bounding boxes over CRs, as doing so vastly reduces the storage requirements (from TBs of raw data to 4 MB in this case), provides a succinct summary of the network congestion state, and eases visualization.

We validate the methodology for determining the parameters of the region-based segmentation algorithm and its applicability for *CR* extraction by using synthetic datasets, as described in Appendix D.

4.3 Implementation and Performance

We have implemented the region-extraction algorithm as a modified version of the region growth segmentation algorithm [78] found in the open-source PointCloud Library (PCL) [9] [4]. The tool is capable of performing run-time extraction of *CRs* even for large-scale topologies. Using the

[10] stall thresholds are scaled by 2.55× to represent the color range (0-255) for visualization purposes

Blue Waters dataset, Monet mined *CRs* from each 60-second snapshot of data for 41,472 links in $\sim$7 seconds; Monet was running on a single thread of a 2.0 GHz Intel Xeon E5-2683 v3 CPU with 512 GB of RAM. Thus, on Blue Waters *Monet* can be run at run-time, as the collection interval is much greater than CR extraction time. Since Monet operates on the database, it works the same way whether the data are being streamed into the database or it is operating on historical data.

5 Characterization Results

In this section, we present results of the application of our analysis methodology to five months of data from a large-scale production HPC system (*Blue Waters*) to provide characterizations of *CRs*. Readers interested in understanding traffic characteristics at the link and datacenter-level may refer to a related work [16].

5.1 Congestion Region Characterization

Here we assess and characterize the congestion severity.

CR-level Size and Severity Characterizations: Figure 5a shows a histogram[11] of *CR* sizes in terms of the number of links for each congested *state* (i.e., not including *'Neg'*). Figure 5b show a histogram of the durations of *CRs* across *'Low'*, *'Medium'* and *'High'* congestion levels. These measurements show that unchecked congestion in credit-based interconnects leads to:

- High growth and spread of congestion leading to large *CRs*. The max size of *CRs* in terms of number of links was found to be 41,168 (99.99% of total links), 6,904 (16.6% of total links), and 2,324 (5.6% of total links) across *'Low'*, *'Medium'* and *'High'* congestion levels respectively, whereas the 99th percentile of the[12] *CR* size was found to be 299, 448, and 214 respectively.
- Localized congestion hotspots, i.e., pockets of congestion. *CRs* rarely spread to cover all of the network. The number of *CRs* decreases (see Figure 5a) with increasing size across all severity *states* except for *'Low'* for which we observe increase at the tail. For example, there are $\sim$16,000 CRs in the *'High'* which comprise 128 links but only $\sim$141 CRs of size $\sim$600.
- Long-lived congestion. The *CR* count decreases with increasing duration, however there are many long-lived *CRs*. The 50%ile, 99%ile and max duration of *CRs* across all states were found to be 579 minutes (9.7 hours), 1421 minutes (23.6 hours), and 1439 minutes (24 hours) respectively, whereas the 50%ile, 99%ile and max P_{Ts} of *CRs* was found to be 14%, 46%, and 92%, respectively. *CR* duration did not change significantly across *'Low'*, *'Medium'*, and *'High'*.

CR Evolution and State Probability: Figure 5c shows the transition probabilities of the *CR* states. The percentage in the box next to each *state* shows the percentage of total link-hours[13] spent in that *state*. It can be interpreted as the probability that a link will be congested at a severity *state* at a given time. For example, there is a probability of 0.10% that a link will be in the *'High'*. These measurements show that:

- The vast majority of link-hours (99.3% of total link-hours) on Blue Waters are spent in *'Neg'* congestion. Consideration of a grosser congestion metric, such as the average stall time across the entire network, will not reveal the presence of significant *CRs*.
- Once a *CR* of *'Low'*, *'Medium'* or *'High'* congestion is formed, it is likely to persist (with a probability of more than 0.5) rather than decrease or vanish from the network.

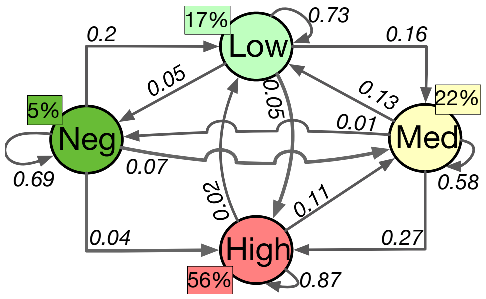


Figure 6: **Network** *congestion evolution captures transition probabilities from one severity* **state** *to another. Percentage numbers in boxes indicates percentage of total system wall clock time spent in that* **state.**

5.2 Network-level Congestion Evolution and Transition Probabilities

In this section, we assess and characterize the overall network congestion severity state. The overall network congestion severity *state* is the *state* into which the highest CR falls. That assignment is independent of the overall distribution of links in each *state*. Figure 6 shows the probabilities that transitions between network *states* will occur between one measurement interval and the next. The rectangular boxes in the figure indicate the fraction of time that the network resides in each *state*. These measurements show the following:

- While each *individual* link of the entire network is most often in a *state* of *'Neg'* congestion, there exists at least one *'High'* CR for 56% of the time. However, *'High'* CRs are small; in Section 5.1, we found that 99th percentile size of *'High'* is 214 links. Thus, the *Blue Waters* network *state* is nearly always non-negligible (95%), with the "High" *state* occurring for the majority of the time.
- There is a significant chance that the current network *state* will persist or increase in severity in the next measurement period. For example, there is an 87% chance that it will stay in a *'High' state*.
- A network *state* is more likely to drop to the next lower *state* than to drop to *'Neg'*.
- Together these factors indicate that congestion builds and subsides slowly, suggesting that it is possible to fore-

[11] plotted as lines and every tenth point marked on the line using a shape for clarity.

[12] We will use %ile to denote percentile in the rest of the paper.

[13] Link-hours are calculated by $\sum$(#links in *Region*) $\times$ (measurement time-window) for each *state*.

cast (within bounds) congestion levels. Combined with proactive localized congestion mitigation techniques and CEMRs, such forecasts could significantly improve overall system performance and application throughput.

5.3 Application Impact of CR

The potential impact of congestion on applications can be significant, even when the percentage of link-hours spent in non-*'Neg'* congested regions is small. While we cannot quantify congestion's impact on all of the applications running on Blue Waters (as we lack ground truth information on particular application runtimes without congestion), we can quantify the impact of congestion on the following:

- Production runs of the NAMD application [77]. The worst-case NAMD execution runtime was 3.4× slower in the presence of high *CRs* relative to baseline runs (i.e., negligible congestion). The median runtime was found be 282 minutes, and hence worst-case runtime was 1.86× slower than the median runtime. This is discussed in more detail in Section 4.1.
- In [16], authors show that benchmark runs of PSDNS [74] and AMR [2] on 256 nodes slowed down by as much as 1.6× even at low-levels of congestion ($5\% < P_{Ts} \leq 15\%$).

To find a upper bound on the number of potentially impacted applications, we consider the applications whose allocations are directly associated with a router in a *CR*. Out of 815,006 total application runs on Blue Waters, over 16.6%, 12.3%, and 6.5% of the unique application runs were impacted by *'Low'*, *'Medium'*, and *'High'* CRs, respectively.

5.4 Congestion Scenarios

In this section, we show how *CRs* manifest under different congestion scenarios: (a) system issues (e.g. changes in system load), (b) network-component failures (e.g. link failures), and (c) intra-application contention. Since the *CRs* are described as bounding boxes with coordinates described in relation to the 3D torus, they can easily be visualized in conjunction with applications' placements at runtime on the torus. *CRs* of *'Neg'* congestion are not shown in the figures.

Congestion due to System Issues: Network congestion may result from contention between different applications for the same network resources. That can occur because of a change in system load (e.g. launches of new applications) or change in application traffic that increases contention on shared links between applications.

Figure 7(i) shows four snapshots, read clockwise, of extracted *CRs*, including size and severity state, for different time intervals during a changing workload. Figure 7(i)(a) shows that *'Low'* (blue) *CRs* when most of the workload consists of multiple instances of MPI-based *NAMD* [77]. The overall network state was thus *'Low'*. The *CRs* remained relatively unchanged for 40 minutes, after which two instances of NAMD completed and *Variant Calling* [37] was launched. Three minutes after the launch, new *CRs* of increased severity

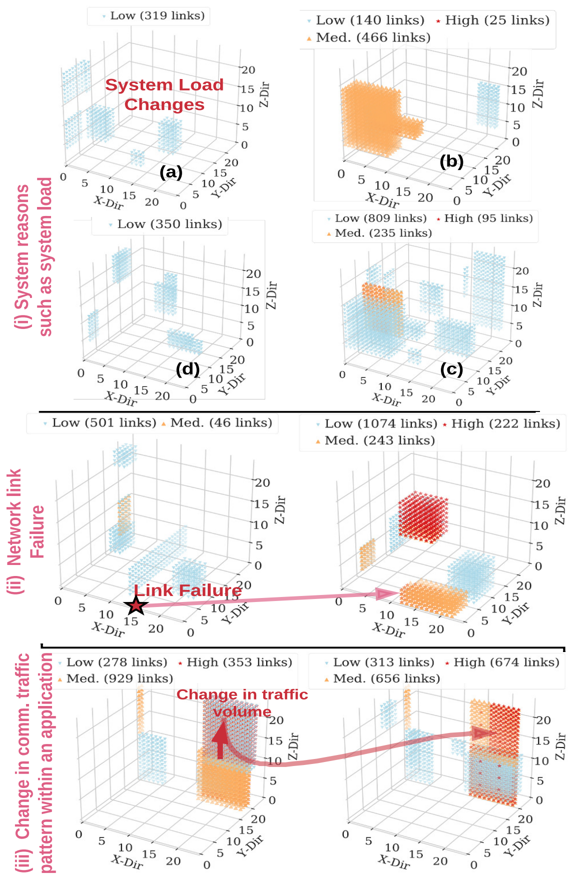


Figure 7: ***Case studies: network congestion is shown due to (i) system issues (such as introduction of new applications), (ii) failures (such as network link failure), and (iii) change in communication pattern within the application.***

occurred (Figure 7(i)(b,c)). The *'High'* (red) [14] and *'Medium'* (orange) severity *CRs* overlapped with the applications.

The increase in the severity of congestion was due to high I/O bandwidth utilization by the *Variant Calling* application. The overall network state remained *'High'* for ∼143 minutes until the *Variant Calling* application completed. At that time, the congestion subsided, as shown in Figure 7(i)(d).

Congestion Due to Network-component Failures: Network-related failures are frequent [55, 68] and may lead to network congestion, depending on the traffic on the network and the type of failure. In [55], the mean time between failures (MTBF) for directional links in Blue Waters was found to be approximately $2.46e06$ link-hours (or 280 link-years). Given the large number of links (41,472 links) on Blue Waters, the expected mean time between failure of a link across the system is about 59.2 hours; i.e., Blue Waters admins can expect one directional-link failure every 59.2 hours.

Failures of directional links or routers generally lead to

[14] not visible and hidden by other regions.

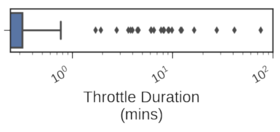
occurrences of *'High' CRs*, while isolated failures of a few switch links (which are much more frequent) generally do not lead to occurrences of significant *CRs*. In this work we found that 88% of directional link failures led to congestion; however, isolated failures of switch links did not lead to significant *CRs* (i.e., had *'Neg' CRs*).

Figure 7(ii) shows the impact of a network blade failure that caused the loss of two network routers and about 96 links (x,y,z location of failure at coordinates (12,3,4) and (12,3,3)). Figure 7(ii)(a) shows the congestion *CRs* before the failure incident and Figure 7(ii)(b) shows the *CRs* just after the completion of the network recovery. Immediately after failure, the stalls increased because of the unavailability of links, requiring the packets to be buffered on the network nodes. The congestion quickly spread into the geometry of nearby applications in the torus. Failure of a blade increased the overall size (in number of links) of *'Low' CRs* by a factor of 2, and of *'Medium' CRs* by a factor of 4.2, and created previously non existent *'High' CRs* with more than 200 links.

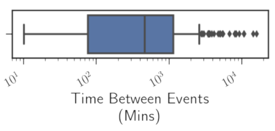
Congestion Due to Intra-Application Issues: Congestion within an application's geometry (intra-application contention) can occur even with TAS. Figure 7(iii) shows congestion *CRs* while the uGNI-based shared memory parallel (SMP) *NAMD* application on more than 2,000 nodes. The application is geometrically mapped on the torus starting at coordinates (15, 18, 0) and ending at coordinates (1, 21, 23) (wrapping around). The congestion *CRs* alternate between the two states shown (state 1 shown in Figure 7(iii)(a), and state 2, shown in Figure 7(iii)(b)) throughout the application run-time because of changes in communication patterns corresponding to the different segments of the NAMD code.

Intra-application contention is less likely to elevate to cause global network issue, unless the links are involved in global (e.g., I/O) routes, or if the resulting congestion is heavy enough to trigger the system-wide mitigation mechanism (see Section 2.2).

Importance of diagnosis: In this section, we have identified three high-level causes of congestion, which we categorize as (a) system issues, (b) network-component failures, and (c) intra-application contention. For each cause, system managers could trigger one of the following actions to reduce/manage congestion. In the case of intra-application congestion, an automated MPI rank remapping tool such as TopoMapping [46], could be used to change traffic flow bandwidth on links to reduce congestion on them. In the case of inter-application congestion (caused by system issues or network failures), a node-allocation policy (e.g., TAS) could use knowledge of congested regions to reduce the impact of congestion on applications. Finally, if execution of an application frequently causes inter-application congestion, then the application should be re-engineered to limit chances of congestion.



(a) Box plot of duration of throttling



(b) Box plot of time between triggers of congestion mitigation events

Figure 8: Characterizing Cray Gemini congestion mitigation events.

6 Using Characterizations: Congestion Response

In this section, we first discuss efficacy of Cray CPEs and then show how our *CR*-based characterizations can be used to inform effective responses to performance-degrading levels of congestion.

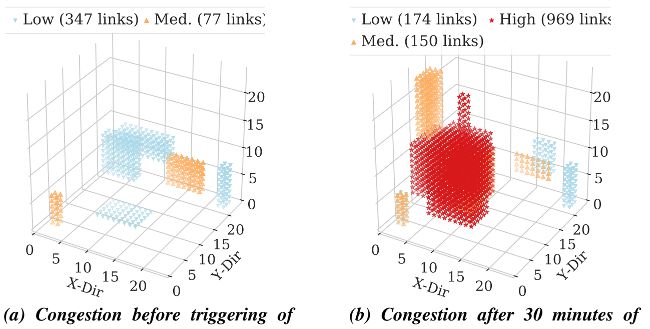
Characterizing Cray *CPEs*: Recall from Section 2 that the vendor-provided congestion mitigation mechanism throttles all NIC traffic injection into the network irrespective of the location and size of the triggering congestion region. This mitigation mechanism is triggered infrequently by design and hence may miss detections and opportunities to trigger more targeted congestion avoidance mechanisms. On Blue Waters, congestion mitigation events are generally active for small durations (typically less than a minute), however, in extreme cases, we have seen them active for as long as 100 minutes. Each throttling event is logged in *netwatch* log files.

We define a *congestion mitigation event (CME)* as a collection of one or more throttling events that were coalesced together based on a sliding window algorithm [63] with a sliding window of 210 seconds, and we use this to estimate the duration of the vendor-provided congestion mitigation mechanisms. Figure 8a and 8b shows a box plot of duration of and time between *CMEs* respectively. The analysis of *CMEs* shows that :

- CMEs were triggered 261 times; 29.8% of which did not alleviate congestion in the system. Figure 9 shows a case where the size and severity of *CRs* increases after a series of throttling events.
- The median time between triggers of CMEs was found to be 7 hours. The distribution of time between events is given in Figure 8b.
- CMEs are generally active for small durations (typically less than a minute), however, in extreme cases, we have seen them active for as long as 100 minutes.
- 8% of the application runs were impacted with over 700 of those utilizing $>$ 100 nodes.

These observations motivate the utility of augmenting the vendor supplied solution of global traffic suppression to manage exceptionally high congestion bursts with our more localized approach of taking action on *CRs* at a higher system-level of granularity to alleviate sources of network congestion.

***CR*-based congestion detection to increase mitigation effectiveness:** *CR* based characterizations can potentially im-



(a) Congestion before triggering of CPE

(b) Congestion after 30 minutes of CPE

Figure 9: A case in which a congestion protection event (CPE) failed to mitigate the congestion

prove congestion mitigation and CEMR effectiveness by more accurately determining which scenarios should be addressed by which mechanisms and by using the identified *CRs* to trigger localized responses more frequently than Cray *CMEs*. That approach is motivated by our discovery (see Section 5.2) that the network is in a *'High'* congestion *state* the majority of the time, primarily because of *CRs* of small size but significant congestion severity.

We define a *Regions Congestion Event (RCE)* as a time-window for which each time instance has at least one region of '*High'* congestion. We calculate it by combining the *CR* evaluations across 5-minute sliding windows. Figure 10 shows boxplots of (a) average credit P_{TS} across all extracted *CRs* during RCEs', (b) average inq P_{TS} across all RCEs', (c) times between RCE, and (d) durations of the RCEs'. These measurements show

- Relative to the vendor-provided congestion mitigation mechanisms, our characterization results in 13× more events (3390 RCEs) upon which we could potentially act.
- Vendor provided congestion mitigation mechanisms trigger on 8% (261 of 3390) of RCEs.
- The average P_{TS} of maximum inq- and credit-stall across all extracted regions present in *RCEs* is quite high, at 33.8% and 27.4%, respectively.
- 25% of 3390 RCEs lasted for more than 30 minutes, and the average duration was found to be approximately an hour.

CRs discovery could also be used for informing congestion aware scheduling decisions. Communication-intensive applications could be preferentially placed to not contend for bandwidth in significantly congested regions or be delayed from launching until congestion has subsided.

7 Using Characterizations: Diagnosing Causes of Congestion

Section 5.4 identifies the root causes of congestion and discusses the the importance of diagnosis. Here we explore that idea to create tools to enable diagnosis at runtime.

7.1 Diagnosis Methodology and Tool

We present a methodology that can provide results to help draw a system manager's attention to anomalous scenarios and potential offenders for further analysis. We can combine system information with the *CR*-characterizations to help diagnose causes of significant congestion. Factors include applications that inject more traffic than can be ejected into the targets or than the traversed links can transfer, either via communication patterns (e.g., all-to-all or many-to-one) or I/O traffic, and link failures. These can typically be identified by observation(s) of anomalies in the data.

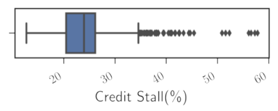
Mining Candidate Congestion-Causing Factors For each congestion Region, CR_i, identified at time T, we create two tables $\mathscr{A}_{CR_i}(T)$ and $\mathscr{F}_{CR_i}(T)$, as described below.

$\mathscr{A}_{CR_i}(T)$ *table*: Each row in $\mathscr{A}_{CR_i}(T)$ corresponds to an application that is within $N_{hops} \leq 3$ hops away from the bounding box of the congestion region CR_i. $\mathscr{A}_{CR_i}(T)$ contains information about the application and its traffic characteristics across seven *traffic features*: (a) application name, (b) maximum read bytes per minute, (c) maximum write bytes per minute, (d) maximum RDMA read bytes per minute, (e) maximum RDMA write bytes per minute, (f) maximum all-to-all communication traffic bytes per minute, and (g) maximum many-to-one communication traffic bytes per minute, where the maximums are taken over the past 30 minutes, i.e., the most recent 30 measurement windows. The list of applications that are within N_{hops} away from congestion region CR_i are extracted from the *workload data*. The measurements for features (a) to (e) are extracted by querying network performance counter data, whereas we estimate the features (f) and (g) are estimated from Network performance counter data by taking several bisection cuts over the application geometry and comparing node traffic ingestion and ejection bytes among the two partitions of the bisection cut.

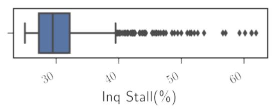
$\mathscr{F}_{CR_i}(T)$ *table*: Each row in $\mathscr{F}_{CR_i}(T)$ corresponds to an application that is within $N_{hops} \leq 3$ away from the congestion boundary of CR_i. $\mathscr{F}_{CR_i}(T)$ contains information about failure events across three *failure features*: (a) failure timestamp, (b) failure location (i.e., coordinates in the torus), and (c) failure type (i.e., switch link, network link, and router failures). Lists of failure events that are within N_{hops} away from congestion region CR_i are extracted from *network failure data*.

Identifying Anomalous or Extreme Factors: The next step is to identify extreme application traffic characteristics or network-related failures over the past 30 minutes that have led to the occurrence of CRs. For each traffic feature in $\mathscr{A}_{CR_i}(T)$, we use an outlier detection method to identify the top k applications that are exhibiting anomalous behavior. The method uses the numerical values of the features listed in table $\mathscr{A}_{CR_i}(T)$. Our analysis framework uses a median-based outlier detection algorithm proposed by Donoho [40] for each CR_i. According to [40], the median-based method is more robust than mean-based methods for skewed datasets. Because *CRs* due to network-related failure events [15] are rare relative to congestion caused by other factors, all failure events that

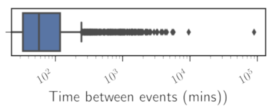
[15]In this paper, we do not consider the effect of lane failures on congestion.



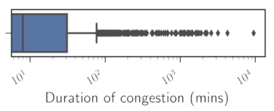
(a) ***Boxplot of average credit stall across extracted congestion events.***



(b) ***Boxplot of average inq-stall across extracted congestion events.***



(c) ***Boxplot of time between congestion events.***



(d) ***Boxplot of duration of congestion.***

Figure 10: Characterization of Regions Congestion Events (RCE).

occur within N_{hops} of CR_i in the most recent 30 measurement windows are marked as anomalous.

Generating Evidence: The last step is to generate evidence for determining whether anomalous factors identified in the previous step are truly responsible for the observed congestion in the *CR*. The evidence is provided in the form of a statistical correlation taken over the most recent 30 measurement time-windows between the moving average stall value of the links and the numerical traffic feature(s) obtained from the data (e.g., RDMA read bytes per minute of the application) associated with the anomalous factor(s). For failure-related anomalous factors, we calculate the correlation taken over the most recent 30 measurement time-windows between the moving average of observed traffic summed across the links that are within N_{hops} away from the failed link(s) and the stall values[16]. A high correlation produces the desired evidence. We order the anomalous factors using the calculated correlation value regardless of the congestion cause. Additionally, we show a plot of stall values and the feature associated with the anomalous factor(s) to help understand the impact of the anomalous factor(s) on congestion.

The steps in this section were only tested on a dataset consisting of the case studies discussed in Section 5.4 and 7 because of lack of ground truth labels on root causes. Creation of labels on congestion causes requires significant human effort and is prone to errors. However, we have been able to generate labels by using the proposed unsupervised methodology, which provides a good starting point for diagnosis.

7.2 Comprehensive Congestion Analysis

In this section, we describe an example use case in which our analysis methodologies were used to detect and diagnose the congestion in a scenario obtained from real data for which the ground truth of the cause was available. The overall steps involved in using our methodologies, included in our *Monet* implementation, for congestion detection and diagnosis are summarized in Figure 11 and described in Section 7. Not all of the steps discussed below are currently automated, but we are working on automating an end-to-end pipeline.

Step 1. Extraction of CR. Figure 11(a) shows that our analysis indicated wide spread high-level congestion across the system (see the left graph in Figure 11(a)). An in-depth analysis of the raw data resulted in identification/detection of congestion regions (see the top-right graph in Figure 11(a)).

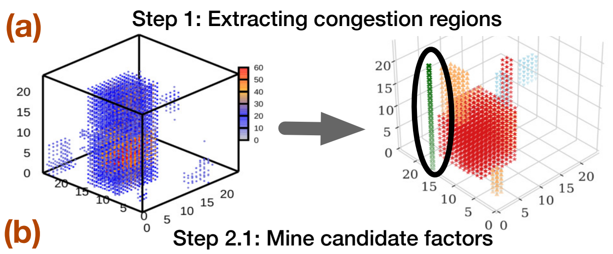
Step 2. Congestion diagnosis. There are 3 steps associated with diagnosing the cause of the congestion.

Step 2.1. Mining candidate factors. To determine the cause of the congestion, we correlated the CR-data with application-related network traffic (for all applications that overlapped with or were near the congestion regions) and network information to generate candidate factors that may have led to congestion. In this example, there were no failures; hence, this analysis generated only application-related candidate factors $\mathscr{A}_{CR_i}$, as shown in Figure 11.

Step 2.2. Identifying anomalous factors. Next, we utilized the application traffic characteristics from candidate factors observed over the last 30 minutes (i.e., many-to-one or all-to-all traffic communication, and file system statistics such as read or write bytes) to identify anomalous factors by using a median-based outlier detection algorithm. In our example, as indicated in Figure 11(b), the offending application was "Enzo" which was running on 32 nodes allocated along the "Z" direction at location (X,Y,Z) = (0,16,16) (indicated by a black circle in Figure 11(a)). At the time of detection, "Enzo" was reading from the file system at an average rate of 4 GB/min (averaged over past 30 minutes and with a peak rate of 70 GB/min), which was 16x greater than the next-highest rate of read traffic by any other application in that time-window. The $\mathscr{A}_{CR_i}(T)$ for RDMA read bytes/min was 70 GB/min. The tool identified the RDMA read bytes/min of the "Enzo" application as the outlier feature. Hence, "Enzo" was marked as the anomalous factor that led to the congestion.

Step 2.3. Generating evidence. Once the potential cause had been established, further analysis produced additional evidence (e.g., distribution and correlation coefficient associated with link stalls in the congestion time window) to validate/verify the diagnosis results produced in Step 2.2. Figure 11(c), in the top graph, shows a plot of the sum of stall rates on all links for all the Gemini routers local to the compute nodes used by the offending application, (i.e., Enzo) (normalized to the total stall rate throughout the duration of the application run). The two peaks (marked) in this top plot correspond to the increase in read bytes (normalized to total read bytes during the application run) shown in the bottom plot. Note that abnormal activity (an excessive amount of traffic to the file system) occurred around 10:10 AM (as shown Figure 11(c)), which was about 20 minutes before the severe congestion developed in the system (seen in Figure 11(a)). A

[16]Increase in traffic near a failed link leads to congestion as shown in Section 5.4.



(b)

Step 2.1: Mine candidate factors

$\mathscr{A}_{CR_i}(T)$

App	RDMA Read	RDMA write	all-to-all	many-to-one
Enzo	70 GiB	~0	~0	1.7 GB/min
Engine_	4/1 GiB/min		~0	~0
...	..	...	...	...

* Traffic measurements max over last 30 minutes (GiB/min)

Step 2.2: Identify anomalous factors

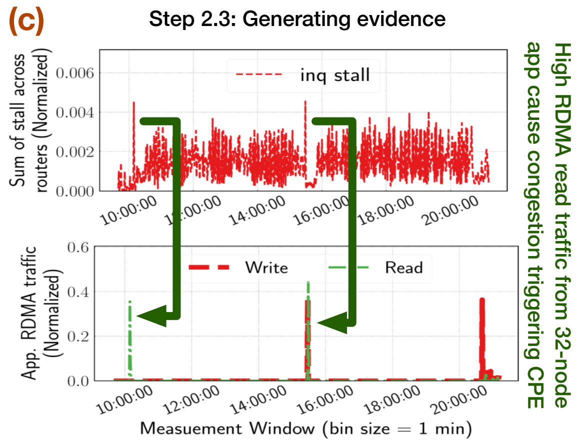


Figure 11: Detection and Diagnosis methodology applied to real-scenario

"Medium" level of congestion was detected in the system spanning a few links (i.e., the congestion region size was small) at the time of the increased read traffic. Thus the cause was diagnosed to be "Enzo". Although, in this example scenario, the Cray congestion mitigation mechanism was triggered, it was not successful in alleviating the network congestion. Instead, the CR size grew over time, impacting several applications. "Enzo" was responsible for another triggering of the congestion mitigation mechanism at 3:20 PM (see the top graph in Figure 11(c)). Monet detected and diagnosed it correctly.

8 Related Work

There is great interest in assessing performance anomalies in HPC systems with the goal of understanding and minimizing application performance variation [25, 86, 86, 88]. Monitoring frameworks such as Darshan [65], Beacon [87] and Kaleidoscope [54] focuses on I/O profiling and performance anomaly diagnosis. Whereas, our work focuses on assessing network congestion in credit-flow based interconnection networks. Typically congestion studies are based on measurements of performance variation of benchmark applications in production settings [25, 88] and/or modeling that assumes steady state utilization/congestion behavior [23, 52, 64, 73], and thus do not address full production workloads.

There are research efforts on identifying hotspots and mitigating the effects of congestion at the application or system-layer (e.g., schedulers). These approaches include (a) use of application's own indirect measures, such as messaging rates [25], or network counters from switch that are accessible only from within an allocation [38, 49, 50, 76], and therefore miss measurements of congestion along routes involving switches outside of the allocation; and (b) use of global network counter data [17, 26, 28, 30], however, these have presented only representative examples of congestion through time or executed a single application on the system [26].

In contrast, this work is the first long-term characterization of high-speed interconnect network congestion of a large-scale production system, where network resources are shared by nodes across disparate job allocations, using global network counters. The characterizations and diagnosis enabled by our work can be used to inform application-level [29] or system-level CEMRs (e.g., use of localized throttling instead of network-wide throttling). Perhaps, the closest work to ours is [22] which is an empirical study of cloud data center networks with a focus on network utilization and traffic patterns, and Beacon [87] which was used on TaihuLight [43] to monitor interconnection network inter-node traffic bandwidth. Like others, these works did not involve generation and characterization of congestion regions, diagnosis of congestion causes, nor a generalized implementation of a methodology for such, however, we did observe some complimentary results in our system (e.g., the existence of hot-spot links, the full bisection bandwidth was not always used, assessment of persistence of congestion in links).

Finally, for datacenter networks, efforts such as ExpressPass [32], DCQCN [89], TIMELY [69] focus on preventing and mitigating congestion at the network-layer whereas efforts such as PathDump [84], SwitchPointer [85], PathQuery [72], EverFlow [90], NetSight [51], LDMS [17] and TPP [53] focus on network monitoring. These approaches are tuned for TCP/IP networks and are orthogonal to the work presented here. Our approach is complementary to these efforts as it enables characterization of congestion regions (hotspots) and identification of congestion causing events.

9 Conclusions and Future Work

We present novel methodologies for detecting, characterizing, and diagnosing network congestion. We implemented these capabilities and demonstrated them using production data from NCSA's 27,648 node, Cray Gemini based, Blue Waters system. While we utilized the scale and data availability of the Blue Waters system to validate our approach, the methodologies presented are generally applicable to other credit-based k-dimensional meshes or toroidal networks. Our future work will involve extending the presented techniques to other network technologies and topologies (see Appendix C).

References

[1] Blue Waters. https://bluewaters.ncsa.illinois.edu.

[2] Charm++ MiniApps . http://charmplusplus.org/benchmarks/#amr.

[3] CLOUD TPU: Train and run machine learning models faster than ever before. https://cloud.google.com/tpu/.

[4] Color-based region growing segmentation. http://pointclouds.org/documentation/tutorials/region_growing_rgb_segmentation.php.

[5] Cray in Azure. https://azure.microsoft.com/en-us/solutions/high-performance-computing/cray/.

[6] Introducing the new HB and HC Azure VM sizes for HPC. https://azure.microsoft.com/en-us/blog/introducing-the-new-hb-and-hc-azure-vm-sizes-for-hpc/.

[7] Monet. https://github.com/CSLDepend/monet.

[8] NVIDIA DGX-1: The Fastest Deep Learning System. https://devblogs.nvidia.com/dgx-1-fastest-deep-learning-system.

[9] Point Cloud Library. http://pointclouds.org.

[10] Post-K Supercomputer Overview. http://www.fujitsu.com/global/Images/post-k-supercomputer-overview.pdf.

[11] Top 500 HPC systems. https://www.top500.org/.

[12] Topology Aware Scheduling. https://bluewaters.ncsa.illinois.edu/topology-aware-scheduling.

[13] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

[14] http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-suite-enterprise-edition.

[15] http://www.nersc.gov/users/computational-systems/edison/.

[16] Anonymized for double blind submission, 2019.

[17] A. Agelastos et al. Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165, 2014.

[18] Yuichiro Ajima, Tomohiro Inoue, Shinya Hiramoto, Shunji Uno, Shinji Sumimoto, Kenichi Miura, Naoyuki Shida, Takahiro Kawashima, Takayuki Okamoto, Osamu Moriyama, et al. Tofu interconnect 2: System-on-chip integration of high-performance interconnect. In *International Supercomputing Conference*, pages 498–507. Springer, 2014.

[19] Yuichiro Ajima, Shinji Sumimoto, and Toshiyuki Shimizu. Tofu: A 6d mesh/torus interconnect for exascale computers. *Computer*, 42(11), 2009.

[20] Yuuichirou Ajima, Tomohiro Inoue, Shinya Hiramoto, and Toshiyuki Shimizu. Tofu: Interconnect for the k computer. *Fujitsu Sci. Tech. J*, 48(3):280–285, 2012.

[21] Robert Alverson, Duncan Roweth, and Larry Kaplan. The Gemini system interconnect. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 83–87. IEEE, 2010.

[22] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[23] Abhinav Bhatele, Nikhil Jain, Yarden Livnat, Valerio Pascucci, and Peer-Timo Bremer. Analyzing network health and congestion in dragonfly-based supercomputers. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2016)*, 2016.

[24] Abhinav Bhatelé and Laxmikant V Kalé. Quantifying network contention on large parallel machines. *Parallel Processing Letters*, 19(04):553–572, 2009.

[25] Abhinav Bhatele, Kathryn Mohror, Steven H Langer, and Katherine E Isaacs. There goes the neighborhood: performance degradation due to nearby jobs. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 41:1–41:12, 2013.

[26] Abhinav Bhatele, Andrew R Titus, Jayaraman J Thiagarajan, Nikhil Jain, Todd Gamblin, Peer-Timo Bremer, Martin Schulz, and Laxmikant V Kale. Identifying the culprits behind network congestion. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 113–122. IEEE, 2015.

[27] Brett Bode, Michelle Butler, Thom Dunning, Torsten Hoefler, William Kramer, William Gropp, and Wen-mei Hwu. The blue waters super-system for super-science. In *Contemporary high performance computing*, pages 339–366. Chapman and Hall/CRC, 2013.

[28] J. Brandt, K. Devine, and A. Gentile. Infrastructure for In Situ System Monitoring and Application Data Analysis. In *Proc. Wrk. on In Situ Infrastructures for Enabling Extreme-scale Analysis and Viz.*, 2015.

[29] J Brandt, K Devine, A Gentile, and Kevin Pedretti. Demonstrating improved application performance using dynamic monitoring and task mapping. In *Cluster Computing, IEEE Int'l Conf. on*. IEEE, 2014.

[30] J. Brandt, E. Froese, A. Gentile, L. Kaplan, B. Allan, and E. Walsh. Network Performance Counter Monitoring and Analysis on the Cray XC Platform. In *Proc. Cray User's Group*, 2016.

[31] Dong Chen, Noel A Eisley, Philip Heidelberger, Robert M Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J Parker. The IBM Blue Gene/Q interconnection network and message unit. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–10. IEEE, 2011.

[32] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 239–252. ACM, 2017.

[33] Cray Inc. Managing Network Congestion in Cray XE Systems. Cray Doc S-0034-3101a Cray Private, 2010.

[34] Cray Inc. Network Resiliency for Cray XE and Cray XK Systems. Cray Doc S-0032-B Cray Private, 2013.

[35] Cray Inc. Managing System Software for the Cray Linux Environment. Cray Doc S-2393-5202axx, 2014.

[36] William J. Dally and Charles L. Seitz. Torus routing chip, June 12 1990. US Patent 4,933,933.

[37] Van der Auwera et al. From fastq data to high-confidence variant calls: the genome analysis toolkit best practices pipeline. *Current protocols in bioinformatics*, pages 11–10, 2013.

[38] M. Deveci, S. Rajamanickam, V. Leung, K. Pedretti, S. Olivier, D. Bunde, U. V. Catalyurek, and K. Devine. Exploiting Geometric Partitioning in Task Mapping for Parallel Computers. In *Proc. 28th Int'l IEEE Parallel and Distributed Processing Symposium*, 2014.

[39] Catello Di Martino, William Kramer, Zbigniew Kalbarczyk, and Ravishankar Iyer. Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 hpc application runs. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 25–36. IEEE, 2015.

[40] David L Donoho. Breakdown properties of multivariate location estimators. Technical report, Technical report, Harvard University, Boston. URL http://www-stat. stanford. edu/˜ donoho/Reports/Oldies/BPMLE. pdf, 1982.

[41] J Enos et al. Topology-aware job scheduling strategies for torus networks. In *Proc. Cray User Group*, 2014.

[42] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, James Reinhard, et al. Cray Cascade: A Scalable HPC System Based on a Dragonfly Network. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 103:1–103:9, 2012.

[43] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, 2016.

[44] Joshi Fullop et al. A diagnostic utility for analyzing periods of degraded job performance. In *Proc. Cray User Group*, 2014.

[45] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the i/o of hpc applications under congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1013–1022. IEEE, 2015.

[46] Juan J. Galvez, Nikhil Jain, and Laxmikant V. Kale. Automatic topology mapping of diverse large-scale parallel applications. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 17:1–17:10, New York, NY, USA, 2017. ACM.

[47] Pedro Javier García, Francisco J Quiles, Jose Flich, Jose Duato, Ian Johnson, and Finbar Naven. Efficient, scalable congestion management for interconnection networks. *IEEE Micro*, 26(5):52–66, 2006.

[48] Paul Garrison. Why is Infiniband Support Important? `https://www.nimbix.net/why-is-infiniband-support-important/`.

[49] R. Grant, K. Pedretti, and A. Gentile. Overtime: A tool for analyzing performance variation due to network interference. In *Proc. of the 3rd Workshop on Exascale MPI*, 2015.

[50] T. Groves, Y. Gu, and N. Wright. Understanding Performance Variability on the Aries Dragonfly Network. In *IEEE Int'l Conf. on Cluster Computing*, 2017.

[51] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know

what your packet did last hop: Using packet histories to troubleshoot networks. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 71–85, 2014.

[52] N. Jain, A. Bhatele, X. Ni, N. J. Wright, and L. V. Kalè. Maximizing Throughput on A Dragonfly Network. In *Proc. Int'l Conference on High Performance Computing, Networking, Storage and Analysis*, 2014.

[53] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 3–14. ACM, 2014.

[54] Saurabh Jha, Shengkun Cui, Tianyin Xu, Jeremy Enos, Mike Showerman, Mark Dalton, Zbigniew T Kalbarczyk, William T Kramer, and Ravishankar K Iyer. Live forensics for distributed storage systems. *arXiv preprint arXiv:1907.10203*, 2019.

[55] Saurabh Jha, Valerio Formicola, Catello Di Martino, Mark Dalton, William T Kramer, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Resiliency of HPC Interconnects: A Case Study of Interconnect Failures and Recovery in Blue Waters. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[56] Saurabh Jha, Archit Patke, Jim M. Brandt, Ann C. Gentile, Mike Showerman, Eric Roman, Zbigniew T. Kalbarczyk, William T. Kramer, and Ravishankar K. Iyer. A study of network congestion in two supercomputing high-speed interconnects. *CoRR*, abs/1907.05312, 2019.

[57] Saurabh Jha, Archit Patke, Mike Showerman, Jeremy Enos, Greg Bauer, Zbigniew Kalbarczyk, Ravishankar Iyer, and William Kramer. Monet - Blue Waters Network Dataset. https://bluewaters.ncsa.illinois.edu/monet-bw-net-data/, 2019.

[58] Matthew D Jones, Joseph P White, Martins Innus, Robert L DeLeon, Nikolay Simakov, Jeffrey T Palmer, Steven M Gallo, Thomas R Furlani, Michael Showerman, Robert Brunner, et al. Workload Analysis of Blue Waters. *arXiv preprint arXiv:1703.00924*, 2017.

[59] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable Dragonfly topology. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 77–88. IEEE, 2008.

[60] William Kramer, Michelle Butler, Gregory Bauer, Kalyana Chadalavada, and Celso Mendes. Blue waters parallel i/o storage sub-system. *High Performance Parallel I/O*, pages 17–32, 2015.

[61] HT Kung, Trevor Blackwell, and Alan Chapman. Credit-based flow control for atm networks: credit update protocol, adaptive credit allocation and statistical multiplexing. In *ACM SIGCOMM Computer Communication Review*, volume 24, pages 101–114. ACM, 1994.

[62] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[63] T.-T.Y. Lin and D.P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *Reliability, IEEE Transactions on*, 39(4):419–432, 1990.

[64] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114. ACM, 2016.

[65] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. A multiplatform study of i/o behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 33–44. ACM, 2015.

[66] Catello Di Martino, Saurabh Jha, William Kramer, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Logdiver: a tool for measuring resilience of extreme-scale systems and applications. In *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*, pages 11–18. ACM, 2015.

[67] Celso L. Mendes, Brett Bode, Gregory H. Bauer, Jeremy Enos, Cristina Beldica, and William T. Kramer. Deploying a large petascale system: The blue waters experience. *Procedia Computer Science*, 29(0):198 – 209, 2014. 2014 International Conference on Computational Science.

[68] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407. ACM, 2018.

[69] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. Timely: Rtt-based congestion control for the datacenter. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 537–550. ACM, 2015.

[70] Hiroyuki Miyazaki, Yoshihiro Kusano, Naoki Shinjou, Fumiyoshi Shoji, Mitsuo Yokokawa, and Tadashi Watanabe. Overview of the k computer system. *Fujitsu Sci. Tech. J*, 48(3):302–309, 2012.

[71] Misbah Mubarak, Philip Carns, Jonathan Jenkins, Jianping Kelvin Li, Nikhil Jain, Shane Snyder, Robert Ross, Christopher D Carothers, Abhinav Bhatele, and Kwan-Liu Ma. Quantifying I/O and communication traffic interference on dragonfly networks equipped with burst buffers. In *Cluster Computing, 2017 IEEE Int'l Conf. on*, pages 204–215. IEEE, 2017.

[72] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. Compiling path queries. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 207–222, 2016.

[73] Vikram Nathan, Srinivas Narayana, Anirudh Sivaraman, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Demonstration of the marple system for network performance monitoring. In *Proceedings of the SIGCOMM Posters and Demos*, pages 57–59. ACM, 2017.

[74] National Center for Supercomputing Applications. SPP-2017 Benchmark Codes and Inputs. `https://bluewaters.ncsa.illinois.edu/spp-benchmarks`.

[75] Zhengbin Pang, Min Xie, Jun Zhang, Yi Zheng, Guibin Wang, Dezun Dong, and Guang Suo. The TH express high performance interconnect networks. *Frontiers of Computer Science*, 8(3):357–366, 2014.

[76] K. Pedretti, C. Vaughan, R. Barrett, K. Devine, and S. Hemmert. Using the Cray Gemini Performance Counters. In *Proc. Cray User's Group*, 2013.

[77] James C Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D Skeel, Laxmikant Kale, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of computational chemistry*, 26(16):1781–1802, 2005.

[78] T Rabbani, F.A. van den Heuvel, and George Vosselman. Segmentation of point clouds using smoothness constraint. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 36, 01 2006.

[79] SchedMD. Slurm scontrol. `https://slurm.schedmd.com/scontrol.html`.

[80] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10435–10444, 2018.

[81] David Skinner and W. Kramer. Understanding the causes of performance variability in hpc workloads. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 137–149, 2005.

[82] Hari Subramoni, Sreeram Potluri, Krishna Kandalla, B Barth, Jérôme Vienne, Jeff Keasler, Karen Tomko, K Schulz, Adam Moody, and Dhabaleswar K Panda. Design of a scalable InfiniBand topology service to enable network-topology-aware placement of processes. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 70:1–70:12, 2012.

[83] Yanhua Sun, Gengbin Zheng, Laximant V Kale, Terry R Jones, and Ryan Olson. A ugni-based asynchronous message-driven runtime system for cray supercomputers with gemini interconnect. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 751–762. IEEE, 2012.

[84] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 233–248, 2016.

[85] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 453–456, 2018.

[86] Ozan Tuncer, Emre Ates, Yijia Zhang, Ata Turk, Jim Brandt, Vitus J Leung, Manuel Egele, and Ayse K Coskun. Diagnosing performance variations in hpc applications using machine learning. In *International Supercomputing Conference*, pages 355–373. Springer, 2017.

[87] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, et al. End-to-end i/o monitoring on a leading supercomputer. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 379–394, 2019.

[88] Xu Yang, John Jenkins, Misbah Mubarak, Robert B Ross, and Zhiling Lan. Watch out for the bully! job interference study on dragonfly network. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 750–760. IEEE, 2016.

[89] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming

Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.

[90] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 479–491. ACM, 2015.

A HPC Interconnect Background

Here we briefly give an overview of HPC interconnects and dive deeper into the details of torus networks.

A.1 Interconnection Networks

An interconnection network is a programmable system that transports data between terminals. The main design aspects of interconnection networks are (1) topology, (2) routing, (3) flow control, and (4) recovery. Topology determines the connection between compute nodes and network nodes (routers, switches, etc.). Routing, flow control, and recovery heavily depend on the topology of the interconnection system. The most widely used topologies in high-performance computing (HPC) are (1) Fat-Tree (e.g. Summit [13]), (2) DragonFly (e.g., Edison [15]), and (3) Torus (e.g., Blue Waters [67]).

A.2 Torus Networks

Torus networks can support $N = k^n$ nodes which are arranged in a *k-ary n-cube* grid (i.e., nodes are arranged in regular n-dimensional grid with k nodes in each dimension). In the case of Blue Waters, $n = 3$. In torus networks, each node serves simultaneously as an input terminal, output terminal and switching node of the network. Torus networks are regular (i.e., all nodes have the same degree) and are also edge-symmetric (useful for load-balancing). Torus networks are very popular for exploiting physical locality between communicating nodes, providing low latency and high throughput. However, the average hop count to route packets to a random node is high compared with other network topolgoies such as Fat-Tree or DragonFly. On the other hand, extra hop counts provide path diversity, which is required for building fault-tolerant architecture.

Routing involves selection of the path from the source node (src) to destination node (dst) among many possible paths in a given topology. In torus networks, routing is done through the directional-order routing algorithm. Directional-order routing does the following:

- Routes the packet in X+/-, Y+, or Z+ until the dimension is resolved,
- Routes the packet in Y+/- or Z+ until the Y dimension is resolved, and
- Routes the packet in Z+/- until the Z dimension is resolved, at which point the packet must have arrived at its destination.

B Workload Information

On Blue Waters, all jobs execute in non-shared mode, without any co-location with another job on the same compute node. Users can submit batch or interactive jobs using Moab/Torque [14] and configure several parameters for job resource request such as: (i) number of nodes, (ii) the number of cores, and (iii) the system walltime (i.e., requested clock time for the job). Blue Waters puts a 48-hour walltime restriction. Blue Waters uses Integrated System Console (ISC) [44] to parse and store the job records and its associated metrics (performance and failure) in its database.

In [39], Di Martino *et al.* provided detailed characterization of more than 5 million HPC application runs completed during the first 518 production days of Blue Waters. However, for completeness, this section provides the workload characteristics of the jobs running on Blue Waters during our study period. Due to loss of data caused by a failure, we do not have workload information for Jan 2017 and hence the workload data shown here is from Feb 2017 - July 2017. During our study period, 2,219k jobs were executed by 467 unique users. We characterize the job characteristics in terms of i) job type, and ii) job size.

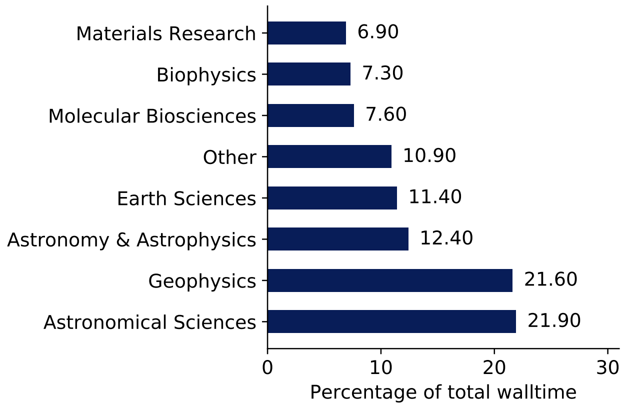
B.1 Job type

Blue Waters workload is predominantly composed of scientific applications. The most prolific scientific fields are summarized in Figure 12a in terms of node-hours. The top scientific discipline during our study period was 'Astronomical Sciences' (21.9%). However, the top scientific disciplines changes over time based on resource allocation awards given by US National Science Foundation and University of Illinois.

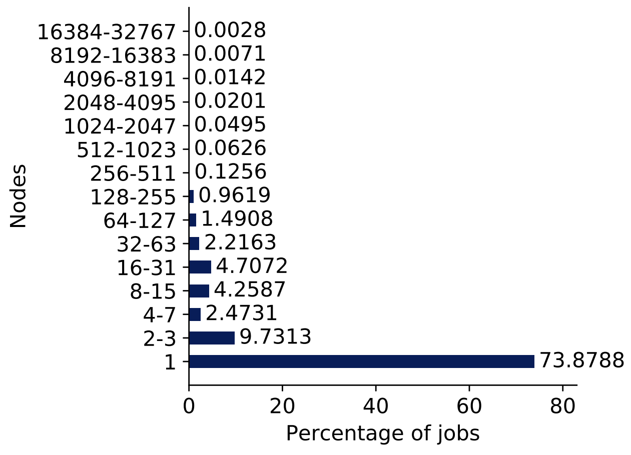
Definition 4 Node-seconds: *is the product of the number of nodes and the wallclock time (in seconds) used by a job. The metric captures the scale of the job's execution across space and time.*

B.2 Job Size

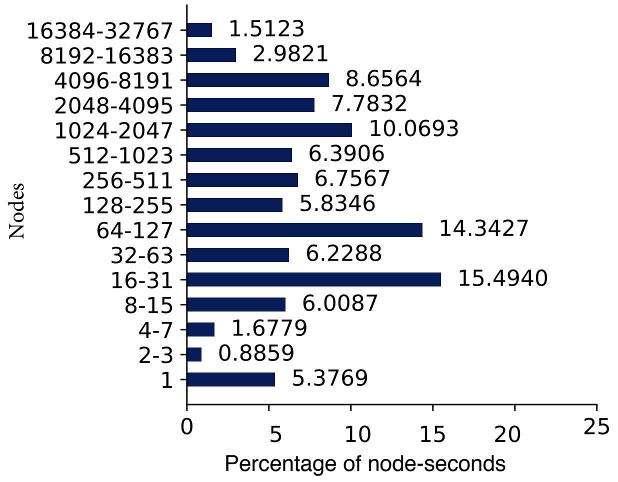
Figure 13b shows a bar plot summarizing relationship between percentage by node-seconds (see Def. 4) and percentage of jobs, whereas Fig. 13a shows a bar plot summarizing relationship between number of nodes and percentage of jobs. 74% of the jobs are single-node jobs. However, these jobs contribute *only* 5% by node-seconds. The large-scale jobs by number are small, they contribute to 94% of the total node-seconds.



(a) Breakdown of wallclock time of scientific application domains



(a) Percentage of jobs by node count



(b) Percentage of node-seconds

Figure 13: Characteristics of jobs running on Blue Waters.

C Existence of Congestion Hotspots and Regions in DragonFly Interconnect

Here we characterize hotspot links on Edison [15], a 2.57 petaflops production system, to showcase continued existence of network congestion problems on a current state of the art network interconnect. Edison uses Cray Aries interconnect which is based on DragonFly topology and uses adaptive routing [59]. We use one week of LDMS data that was collected from Edison at one second interval, and amounts to 7.7 TB. Our analyses shows 1) presence of long duration hotspot links, and 2) performance variability on a current state of the art network interconnect.

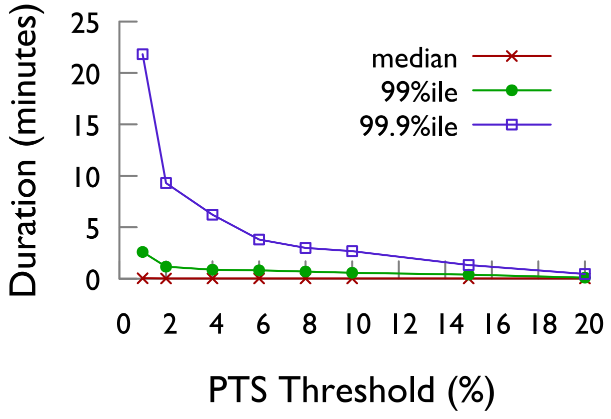


Figure 14: Distribution of hotspot link duration in Edison [replicated from [56]]

Figure 14 characterizes the median, 99%ile and 99.9%ile duration of the hotspot links by generating the distribution of the duration for which a link persists to be in congestion at $P_{Ts} \geq P_{Ts}$ Threshold. While the 99.9%ile hotspot duration is an order of magnitude lesser compared to the observed results in Gemini (see Figure. 3), which can be explained by the low diameter topology and use of congestion-aware routing policies in Aries. The duration of hotspot is longer than a minute for congestion thresholds less than 15% P_{Ts}. More details on characterization for Edison can be found in [56].

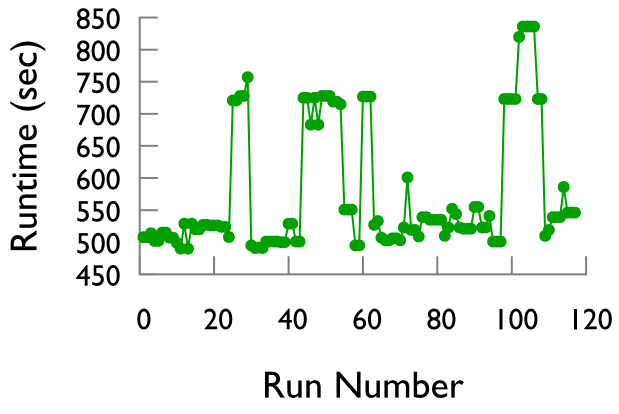


Figure 15: Variation of MILC runtime on Edison

Although the hotspot link duration has significantly decreased, the performance variation due to congestion continues to be a problem. For example, we observed significant performance variation of up to 1.67× compared to baseline for MILC application [25] on Edison (see Figure 15). MILC is a communication-heavy application susceptible to congestion on the interconnect. The reason for slowdown of the application can be attributed to presence of congestion in the links which rapidly evolves (i.e., a link may not be continuously congested but there is always groups of congested links). Preliminary analysis of network congestion counters obtained from Edison [15] suggests existence of congestion regions that evolve rapidly. Our future work will focus on applying and extending our methodology to other interconnects technologies that use different topology other than torus (e.g., Fat-Tree or DragonFly) or use adaptive-routing (e.g., UGAL [59]). In emerging network technologies, vastly more network performance counters are available that can be used for detecting congestion and hence there is an increased need for algorithmic methods for discovering and assessing congestion at system and application-level.

D Validation of Congestion Regions Generated by Region Segmentation

Validation of the region segmentation algorithm was done by inspecting visualizations of both the unsegmented and segmented congestion data. We also generated synthetic congestion data and evaluated our algorithm's performance on it as a sanity check.

D.1 Results Analysis Discussion

As we do not have any ground truth for our clustering algorithm and the credit and inq stall on each link widely varies across the system with time (as discussed in previous subsection), we attempted some sanity checks in order to validate that the algorithm produced a sensible clustering of the data. To facilitate this, we implemented visualization tools for visualizing both the raw, unsegmented data, as well as the final, segmented data. We then ran our algorithm on the congestion data that was recorded at times when we knew there were congestion events (e.g. when Cray congestion protection events were triggered), as well as at multiple randomly sampled timepoints. We then visualized both sets of data and manually inspected the regions generated, checking visually to see if they lined up with the visualization of the raw congestion data. For samples that we inspected, the algorithm worked well for segmenting the data.

As a further test, we generated random congestion data following a simplified model and scored our algorithm's effectiveness at classifying those data. The data was created by randomly generating regions of congestion in a 24 x 24 x 24 cube representing the 3D torus of the Blue Waters Gemini interconnect. Each congestion region was created by randomly generating a) a cuboid in which each dimension was between 3 and 9 links inclusive, b) a random stall value s between 20% and 50%, and c) a random integer from $\{0, 1\}$. Depending on the value of the random integer, s was added to the credit-stall or inq-stall of all the links in the cuboid. Finally, after all regions were added random Gaussian noise with $(\mu, \sigma) = (0, 2.5)$ was added to both the credit- and inq-stalls of all the links in the cube to simulate small variances in the stall values of each link.

We then ran our algorithm on 100 samples, each with a random number of regions (from 1 to 8 inclusive), and assigned a score as well as calculating the precision and recall for that sample. We scored the match as follows: for a single sample, let $A_i, i = 1 \cdots n$ be the actual regions and $B_i, i = 1 \cdots m$ be the regions the algorithm produced. A single sample was then assigned the score $(\frac{1}{n}\sum_{i=1}^{n}\frac{|A_i \cap B_{j_i}|}{|A_i \cup B_{j_i}|})(\frac{n}{\max(n,m)})$, where $|A_i|$ represents the number of links in the enclosed region, and the B_{j_i} are the regions that "best" overlap their respective A_i. The B_{j_i} are chosen by going through the regions A_i in order from smallest to largest, and choosing j_i such that B_{j_i} has the largest possible overlap with A_i. Regions created by segmentation that have both inq- and credit-stall $< 5\%$ are considered insignificant and were excluded from this processing.

This scoring assigns a score of 1 to a perfect match, which degrades to 0 when a) the mismatch between the true and the matching generated region increases, or when b) the algorithm generates more regions than true regions.

Based on that scoring, the algorithm achieved an average score of 0.81 (maximum 1.0) with parameters $(\theta_p = 12, \theta_r = 8, \theta_d = 2, \theta_s = 20)$ over 100 samples, with an average precision of 0.87 and an average recall of 0.89.

D.2 Summary

While it is not possible to fully validate the efficacy of our segmentation algorithm, the synthetic datasets generated give us a degree of confidence that the algorithm does the right thing on simple models of congestion that satisfy our assumptions (i.e. that congestion tends to spread locally) and work well with noise. The comparison between the datasets before and after the segmentation suggests that algorithm does still work reasonably well in practice, on real datasets.

The region segmentation algorithm was applied to 5 months of production data collected as part of the operational environment of a system. The data was obtained from Blue Waters through various tools and counters(e.g., network performance counters, link-aggregated data, scheduler and log files) and hence is reliant on the correctness. The reliability of the system software for synchronized data collection at regular intervals (LDMS) is dependent on the system operation staff which supports it and performance details for this software are available in its documention.

Acknowledgment

We thank Jeremy Enos and Brett Bode (NCSA), Eric Roman (NERSC), anonymous reviewers of NSDI 2019 and our shepherd Lalith Suresh for insightful conversations and/or feedback. We thank K. Atchley and J. Applequist for their help in preparing the manuscript .

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number 2015-02674. This work is partially supported by NSF CNS 15-13051, and a Sandia National Lab contract number 1951381.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications (NCSA).

Sandia National Laboratories (SNL) is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues

Albert Gran Alcoz
ETH Zürich

Alexander Dietmüller
ETH Zürich

Laurent Vanbever
ETH Zürich

Abstract

Push-In First-Out (PIFO) queues are hardware primitives which enable programmable packet scheduling by providing the abstraction of a priority queue at line rate. However, implementing them at scale is not easy: just hardware designs (not implementations) exist, which support only about 1k flows.

In this paper, we introduce SP-PIFO, a programmable packet scheduler which closely approximates the behavior of PIFO queues using strict-priority queues—*at line rate, at scale, and on existing devices*. The key insight behind SP-PIFO is to dynamically adapt the mapping between packet ranks and available strict-priority queues to minimize the scheduling errors with respect to an ideal PIFO. We present a mathematical formulation of the problem and derive an adaptation technique which closely approximates the optimal queue mapping without any traffic knowledge.

We fully implement SP-PIFO in P4 and evaluate it on real workloads. We show that SP-PIFO: (i) closely matches PIFO, with as little as 8 priority queues; (ii) scales to large amount of flows and ranks; and (iii) quickly adapts to traffic variations. We also show that SP-PIFO runs at line rate on existing hardware (Barefoot Tofino), with a negligible memory footprint.

1 Introduction

Until recently, packet scheduling was one of the last bastions standing in the way of complete data-plane programmability. Indeed, unlike forwarding whose behavior can be adapted thanks to languages such as P4 [7] and reprogrammable hardware [2], scheduling behavior is mostly set in stone with hardware implementations that can, at best, be configured.

To enable programmable packet scheduling, the main challenge was to find an appropriate abstraction which is flexible enough to express a wide variety of scheduling algorithms and yet can be implemented efficiently in hardware [22]. In [23], Sivaraman et al. proposed to use Push-In First-Out (PIFO) queues as such an abstraction. PIFO queues allow enqueued packets to be pushed in arbitrary positions (according to the packets rank) while being drained from the head.

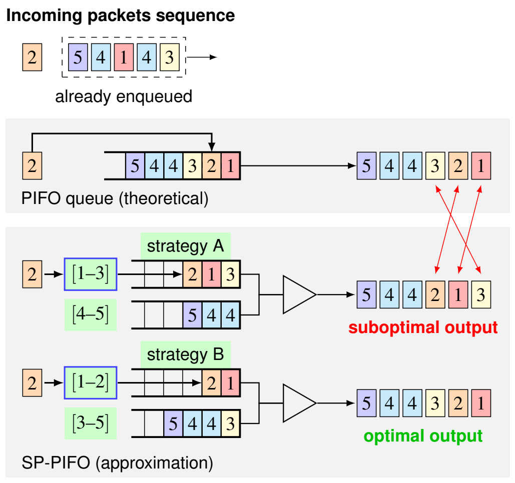


Figure 1: SP-PIFO approximates the behavior of PIFO queues by adapting how packet ranks are mapped to priority queues.

While PIFO queues enable programmable scheduling, implementing them in hardware is hard due to the need to arbitrarily sort packets at line rate. [23] described a possible hardware design (not implementation) supporting PIFO on top of Broadcom Trident II [1]. While promising, realizing this design in an ASIC is likely to take years [6], not including deployment. Even ignoring deployment considerations, the design of [23] is limited as it only supports ~1000 flows and relies on the assumption that the packet ranks increase monotonically within each flow, which is not always the case.

Our work In this paper, we ask whether it is possible to approximate PIFO queues at scale, in existing programmable data planes. We answer positively and present SP-PIFO, an adaptive scheduling algorithm that closely approximates PIFO behaviors on top of widely-available Strict-Priority (SP) queues. The key insight behind SP-PIFO is to dynamically adapt the mapping between packet ranks and SP queues in order to minimize the amount of scheduling mistakes relative to a hypothetical ideal PIFO implementation.

Example First, we provide an intuition how SP-PIFO approximates PIFO behaviors using SP queues in Fig. 1. The example illustrates the scheduling behavior of two SP-PIFO systems which receive the input packet sequence $\boxed{2}\boxed{5}\boxed{4}\boxed{1}\boxed{4}\boxed{3}$. By convention, we write the first packet being enqueued on the far-right ($\boxed{3}$) and the last one on the far-left ($\boxed{2}$). Similarly to [23], we also consider that lower-rank packets have higher priority (and use corresponding color codes). The figure illustrates the scheduling decision of each system for the sixth packet ($\boxed{2}$), assuming the first 5 have been enqueued already.

A PIFO queue always schedules incoming packets perfectly, leading to the sorted output $\boxed{5}\boxed{4}\boxed{4}\boxed{3}\boxed{2}\boxed{1}$. In contrast, the quality of the scheduling of a SP-PIFO scheme depends on: (i) the number of SP queues available (here, two); and (ii) the mapping of packet ranks to those queues. Fig. 1 illustrates two such mapping strategies. Strategy A maps ranks 1–3 (resp. 4–5) to the highest (resp. lowest) SP queue, while Strategy B maps ranks 1–2 (resp. 3–5) to the highest (resp. lowest) SP queue. We see that Strategy B is capable of perfectly sorting the input sequence, i.e. it behaves like a perfect PIFO queue. In contrast, Strategy A leads to sub-optimal packet inversions, e.g. $\boxed{1}$ is incorrectly scheduled after $\boxed{3}$.

Insights The key challenge in SP-PIFO is to design adaptation strategies that can: (i) closely approximate PIFO behavior; and (ii) be implemented in programmable data planes. These are hard challenges as the best mapping strategy depends on the traffic mix and the actual ranks being enqueued, both of which can change on a per-packet basis.

SP-PIFO approximates the best mapping strategy by dynamically shifting the ranks mapped to each queue to reduce the scheduling mistakes it observes in real time. We show that SP-PIFO's adaptation strategy achieves almost the same performance as provably-correct adaptation strategies while being implementable in programmable data planes.

Performance We use SP-PIFO to implement a wide variety of scheduling objectives ranging from minimizing flow completion times to achieving max-min fairness. For all cases, we show that SP-PIFO achieves performance on-par with the state-of-the-art. We also demonstrate that SP-PIFO runs at line rate on existing programmable hardware.

Contributions Our main contributions are:

- A novel approach for approximating PIFO queues using strict-priority queues (§3).
- An adaptation algorithm which dynamically adapts the queue mapping according to the network conditions, closely-approximating an optimal scheme (§4).
- An implementation[1] of SP-PIFO in Java and P4 (§5).
- A comprehensive evaluation showing SP-PIFO effectiveness in approximating perfect PIFO behavior with as little as 8 queues and on actual hardware switches (§6).

[1]Available at https://github.com/nsg-ethz/sp-pifo

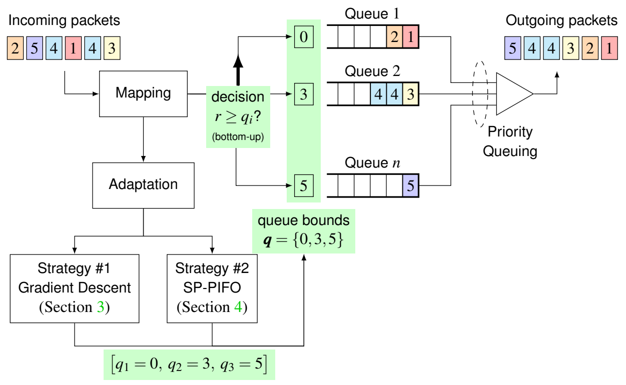


Figure 2: Overview of SP-PIFO data-plane pipeline.

2 Overview

In this section, we provide an informal overview of how SP-PIFO manages to closely approximate PIFO behaviors. At a high level, SP-PIFO is a priority-queuing scheduling discipline (see Fig. 2) which maps incoming packets to n priority queues. SP-PIFO assumes that packets are tagged with a rank indicating the intended scheduling order, with lower ranks being preferred over higher ones. Packets enqueued in a queue are scheduled according to their order of arrival (i.e., First-In First-Out), after *all* packets enqueued in any higher-priority queue have been scheduled. Unlike classical priority-queuing disciplines [20], SP-PIFO dynamically adapts the mapping between the packet ranks and the priority queues according to the observed network conditions. In particular, SP-PIFO adapts the mapping so as to minimize the scheduling "unpifoness", that is, the number of times a higher-rank packet is scheduled *before* an enqueued lower-rank packet. We refer to such scheduling mistakes as *inversions*.

Mapping SP-PIFO maps each incoming packet to queues according to the queue *bounds*. These queue bounds identify, for each queue i, the smallest packet rank that can be enqueued. Whenever a packet is received, SP-PIFO scans the queue bounds bottom-up, starting from the lowest-priority queue, and enqueues the packet in the first queue with a bound smaller or equal to the packet rank. Given a packet with rank $r \in \mathbb{Z}_{\geq 0}$ and n priority queues, let $\boldsymbol{q}$ be the vector of queue bounds $(q_1, \cdots, q_n) \in \mathbb{Z}^n$ such that $0 \leq q_1 \leq q_2 \leq \cdots \leq q_n$. For instance, consider a vector $\boldsymbol{q} = \{0,3,5\}$ indicating the bounds of 3 priority queues, with 0 (resp. 5) indicating the bound of the highest- (resp. lowest-) priority queue. Given $\boldsymbol{q}$, SP-PIFO enqueues packets with rank 2 in the first (highest-priority) queue, packets with rank 3 in the second queue and packets with rank 10 in the third (lowest-priority) queue.

Adaptation "Unpifoness" can be minimized across multiple packets, e.g. by monitoring the rank distribution over periodic time windows and adapting the bounds through a gradient descent, or on a per-packet basis (see Fig. 2). Depending on the characteristics of the rank distribution, the first strategy can provably converge to the optimal mapping. Unfortunately, its requirements exceed the capabilities of existing programmable data planes. SP-PIFO addresses these two limitations: it works for *any* rank distribution, on existing hardware. SP-PIFO dynamically adapts **q** such that the resulting scheduling closely approximates an ideal PIFO queue, minimizing the amount of observed *inversions* by dynamically shifting the ranks mapped to each queue. SP-PIFO operates online, *without* prior knowledge of the incoming packet ranks.

SP-PIFO's adaptation mechanism consists of two stages: a *push-up* stage where future low-rank (i.e. high-priority) packets are pushed to higher-priority queues; and a *push-down* stage where future high-rank (i.e. low-priority) packets are pushed down to lower queues.

Stage 1: Push-up Whenever SP-PIFO enqueues a packet, it updates the corresponding queue bound to the rank of the enqueued packet. Doing so, SP-PIFO aims at ensuring that future lower-ranked packets will not be enqueued in the same queue, but in a more preferred one. Intuitively, SP-PIFO "pushes up" packets with low ranks to the highest-priority queues, where they will drained first. Of course, as the number of queues is finite—and often, much smaller than the number of ranks—this is not always possible, leading to inversions.

Stage 2: Push-down Whenever SP-PIFO detects an inversion in the highest-priority queue (i.e., the packet rank is smaller than the highest-priority queue bound), it decreases the queue bound of *all* queues. Doing so, SP-PIFO ensures that future higher-rank packets will be enqueued in lower-priority queues. Intuitively, after an inversion, SP-PIFO "pushes down" packets with high ranks to the lower-priority queues in order to prevent them from causing inversions in the highest-priority queue. SP-PIFO decreases the queue bounds according to the magnitude of the inversion, i.e. the difference between the packet rank and the corresponding queue bound: the bigger the inversion, the more ranks are pushed down.

Example Fig. 3 illustrates the execution of SP-PIFO with two priority queues when receiving $\boxed{1}\boxed{2}\boxed{5}\boxed{4}\boxed{1}\boxed{4}\boxed{3}$. Without loss of generality, we consider that the queue bounds are initialized to 0. SP-PIFO enqueues the first packet ($\boxed{3}$) in the lowest-priority queue and updates its queue bound to 3. Likewise, SP-PIFO also enqueues the second packet, $\boxed{4}$, in the lowest-priority queue. As its rank (4) is higher than the queue bound (3), it then updates the queue bound to 4.

The same process is applied to the subsequent packets until the second $\boxed{1}$ is encountered, creating an inversion (grayed area in Fig. 3). Indeed, SP-PIFO enqueues $\boxed{1}$ in the highest-priority queue *after* having enqueued $\boxed{2}$. Once the inversion is detected, SP-PIFO adapts the queue bounds to 1 and $5-1=4$, respectively. Observe that if $\boxed{1}$ and $\boxed{2}$ keep arriving, the bound of the lowest-priority queue will decrease, eventually reaching 2. At this point, future $\boxed{1}$ will not experience inversions anymore as they will have a dedicated queue.

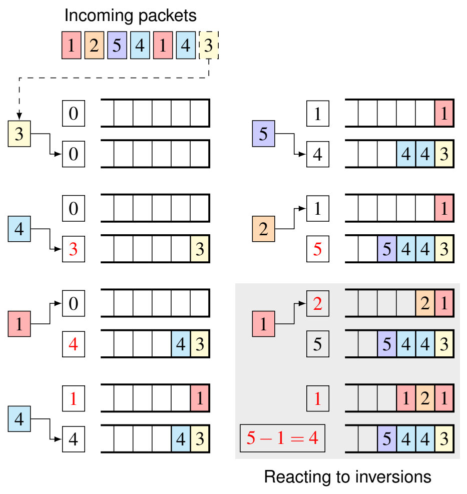


Figure 3: SP-PIFO mapping and adaptation mechanisms.

3 SP-PIFO design

In this section, we describe the theoretical basis supporting the design of SP-PIFO. We first phrase the problem of finding the optimal queue bounds as an empirical risk minimization problem in which a loss function—how "unpifo" the current mapping is—is minimized (§3.1). We then develop an algorithm based on gradient descent which provably converges to the optimal bounds for stable rank distributions (§3.2). We show how the convergence requirements make the algorithm impractical (§3.3). In the following, we present SP-PIFO which relaxes the requirements at the benefit of practicality (§4).

3.1 Problem statement

Let $\mathcal{U} : \mathrm{R}^n \times \mathrm{R}_{\geq 0} \rightarrow \mathrm{R}_{\geq 0}$ be a loss function such that $\mathcal{U}(\boldsymbol{q}, r)$ quantifies the approximation error of scheduling a packet with rank r based on queue bounds $\boldsymbol{q}$ compared to an ideal PIFO queue. Intuitively, a smaller loss equals a better approximation. Note that $\mathcal{U}$ stands for *unpifoness*.

The adaptation goal is to find the optimal queue bounds $\boldsymbol{q}^*$ that minimize the expected loss for all possible ranks. Let $\mathcal{Q}$ be the space of all valid bound vectors and $\mathcal{R}$ the distribution of packet ranks, then the optimal queue bounds $\boldsymbol{q}^*$ are:

$$\boldsymbol{q}^* = \arg\min_{\boldsymbol{q} \in \mathcal{Q}} \mathop{\mathbb{E}}_{r \sim \mathcal{R}} \left[\, \mathcal{U}(\boldsymbol{q}, r) \, \right] \tag{1}$$

Finding $\boldsymbol{q}^*$ directly is intractable though. Indeed, evaluating the expected loss $\mathcal{U}$ is impossible since the distribution of packet ranks $\mathcal{R}$ is unknown. We address this problem by considering the *empirical loss* $\mathcal{U}_{emp}$ observed over a set $\mathcal{D}$ of i.i.d. rank samples. Doing so, we phrase the problem of finding $\boldsymbol{q}^*$ as an *empirical risk minimization* (ERM) problem:

$$\boldsymbol{q}^* = \arg\min_{\boldsymbol{q} \in \mathcal{Q}} \frac{1}{|\mathcal{D}|} \sum_{r \in \mathcal{D}} \mathcal{U}_{emp}(\mathcal{D}, \boldsymbol{q}, r) \tag{2}$$

Evaluating empirical losses For a given rank r, we measure the empirical loss $\mathcal{U}_{emp}$ as the *expected* number of inversions that r would encounter, if the rank distribution $\mathcal{D}$ was scheduled given the queue bounds $\boldsymbol{q}$, weighted by the *cost* that each inversion would cause to the system performance. This cost can be just a constant value, if all inversions are treated the same, or it can measure the magnitude of the inversion (i.e., how big is the difference between ranks causing it). Since r receives inversions only from higher ranks in the distribution, $\mathcal{U}_{emp}$ can be rewritten as:

$$\mathcal{U}_{emp}(\mathcal{D},\boldsymbol{q},r) = \frac{1}{|\mathcal{D}|}\sum_{\substack{r'\in\mathcal{D}\\ r'>r}} \text{cost}_{\boldsymbol{q}}(r',r) \tag{3}$$

Having formulated the adaptation goal as an empirical risk minimization, we aim to solve it by analyzing how changes in $\boldsymbol{q}$ influence the empirical risk, and trying to design an iterative algorithm capable of converging to the minimal risk.

3.2 Gradient-based adaptation algorithm

We first introduce a greedy, gradient-based algorithm, which provably converges to the optimal queue bounds $\boldsymbol{q}^*$ provided that the rank distribution stays constant. The algorithm builds upon the fact that inversions cannot occur between ranks mapped to different priority queues. This allows to instantiate the empirical risk minimization in eq. 2 at a *queue level* by simply adding the individual losses of each queue. Letting $\mathcal{U}(q_i)$ be the loss function corresponding to the queue with bound q_i, this is:

$$\boldsymbol{q}^* = \underset{\boldsymbol{q}\in Q}{\arg\min} \sum_{q_i\in\boldsymbol{q}} \mathcal{U}(q_i) \tag{4}$$

Letting $p_{\mathcal{D}}(r)$ and $p_{\mathcal{D}}(r')$ be the empirical probability of ranks r and r', respectively, both mapped to the queue with bound q_i, we can define the unpifoness of the queue as:

$$\mathcal{U}(q_i) = \sum_{\substack{q_i\le r<q_{i+1}\\ r<r'<q_{i+1}}} p_{\mathcal{D}}(r)\cdot p_{\mathcal{D}}(r')\cdot \text{cost}(r',r) \tag{5}$$

Overview Considering this problem instantiation, the greedy algorithm first computes the rank distribution over a set of k packets before minimizing the expected per-queue unpifoness by incrementing (resp. decrementing) the queue bounds. Specifically, after processing the k-th packet, the greedy algorithm selects, for each queue, the bound that most decreases the overall system unpifoness. Although comparing the performance of all bound combinations is not possible, we introduce an efficient computation mechanism that allows to prune the search space while preserving convergence. We prove the optimality of the algorithm in Appendix A.

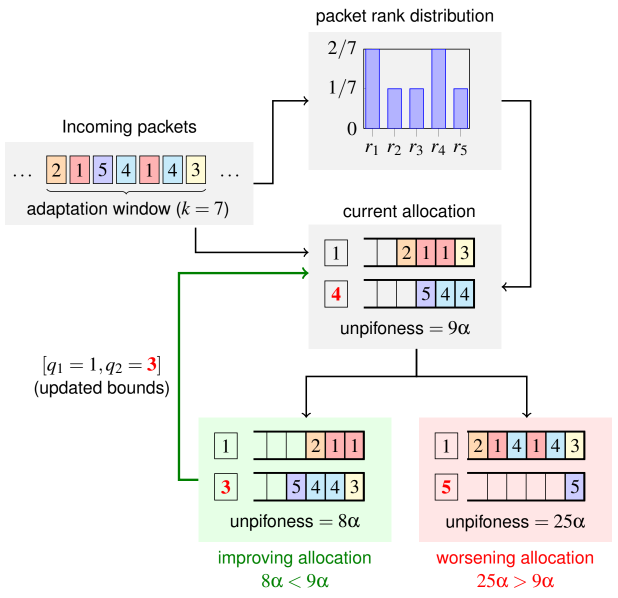


Figure 4: The gradient-based algorithm greedily minimizes the expected *unpifoness*.

Example We illustrate the execution of the algorithm in Fig. 4. We assume a system with two priority queues and assume that the packet sequence 2 1 5 4 1 4 3 is received over and over again. We set the adaptation window k to 7 packets. We initialize the queue bounds to 1 and 4.

The algorithm starts by computing the observed rank distribution after receiving the 7-th packet. Here, it estimates the probability of receiving a packet of rank 1 as $p(1) = 2/7$. Similarly, $p(2) = 1/7$, $p(3) = 1/7$, $p(4) = 2/7$ and $p(5) = 1/7$. It then computes the expected unpifoness that this distribution would have generated with the current queue bounds (eq. 3). For the higher-priority queue, this is $\mathcal{U}_1 = p(1)\cdot p(2)\cdot cost(2,1) + p(1)\cdot p(3)\cdot cost(3,1) + p(2)\cdot p(3)\cdot cost(3,2) = (2/7\cdot 1/7)\cdot(2-1)+(2/7\cdot 1/7)\cdot(3-1)+(1/7\cdot 1/7)\cdot(3-2)$. This equation can be simplified to $\mathcal{U}_1 = 7\alpha$ where $\alpha = (1/7\cdot 1/7)$. Similarly, $\mathcal{U}_2 = p(4)\cdot p(5)\cdot cost(5,4) = 2\alpha$, adding up a total of $\mathcal{U} = 9\alpha$.

Next, the algorithm compares the expected unpifoness that would be obtained if the queue bound was incremented (gradient up) or decremented (gradient down) and adapts the queue bound in the direction resulting in the biggest decrease of unpifoness.

Gradient up Incrementing q_2 from 4 to 5 means that only rank $\{5\}$ would be mapped to the lower-priority queue. The resulting unpifoness is $\mathcal{U} = 25\alpha$. The higher unpifoness (25α instead of 9α) indicates that, by incrementing q_2, the system gets further away from the PIFO behavior. Note that the increase in unpifoness comes from the higher-priority queue as rank $\{5\}$ gets an exclusive queue.

Gradient down In contrast, the system unpifoness reduces from 9α to 8α when decrementing q_2 from 4 to 3. Indeed, $\mathcal{U}_1 = p(1) \cdot p(2) \cdot cost(2,1) = 2\alpha$, and $\mathcal{U}_2 = p(3) \cdot p(4) \cdot cost(4,3) + p(3) \cdot p(5) \cdot cost(5,3) + p(4) \cdot p(5) \cdot cost(5,4) = 6\alpha$, adding up to $\mathcal{U} = 8\alpha$. As such, the adaptation mechanism updates the queue bound: $q_2 = 3$.

The above process repeats every 7-th packet, estimating the rank distribution before greedily adapting the queue bounds.

3.3 Limitations

While the adaptation algorithm described above provably converges to the optimal mapping (see A.1), two key limitations make it impractical. First, it is not currently implementable in existing programmable data planes due to resource constraints. Second, the algorithm only converges for stable rank distributions, which is rarely the case, and its convergence time directly depends on the distribution size, which can be large. We explain how to overcome these limitations in §4.

Hardware restrictions Monitoring the rank distributions over periodic adaptation windows requires a high amount of memory and computational resources, both of which are scarce in current programmable data planes. In particular, implementing the greedy algorithm in hardware (see A.2) requires to: (i) store the value of each queue bound; (ii) compute the current unpifoness; and (iii) estimate the unpifoness obtained by incrementing or decrementing each queue bound. As we explain in A.3, the amount of resources required to run the algorithm on a practical number of queues (8 queues or more) exceeds the capabilities of current switch designs.

Convergence In A.4, we study the performance of the gradient-based algorithm and analyze the effects on convergence when the adaptation window, the number of queues, and the rank range is modified. We show that, for the algorithm to converge, the rank distribution needs to be stable in time. However, this is unrealistic in most practical scenarios where not only the rank distribution *is unknown* but also varies through time (e.g., virtual times in fair-queuing schemes).

4 Our approach: SP-PIFO

We now present SP-PIFO, an approximation of the gradient-based adaptation algorithm (§3.2) which is implementable in existing data planes and rapidly adapts to varying rank distributions. SP-PIFO substitutes the gradient computation by a simpler adaptation process which minimizes the probability of inversions *per packet*, rather than per k-packets.

In the following, we first show how to instantiate the empirical risk minimization problem (eq. 2) at the packet level and describe how SP-PIFO solves it (§4.1). We then systematically characterize how SP-PIFO handles inversions (§4.2).

4.1 Per-packet adaptation algorithm

The SP-PIFO adaptation algorithm (alg. 1) is based on two competing stages that act in opposing direction. We show that this combination manages to strike a balance in the number of inversions observed by all queues, resulting in a good PIFO approximation. In the following, we first show how to phrase the empirical risk minimization problem at the per-packet level before describing both mechanisms.

Problem statement In contrast to §3.2, we aim at minimizing the cost generated by scheduling each individual packet. Formally, we aim to find the optimal bound vector $\boldsymbol{q}^*$ that minimizes the unpifoness for all enqueued packets $\mathcal{P}$:

$$\boldsymbol{q}^* = \arg\min_{\boldsymbol{q} \in \mathcal{Q}} \mathcal{U}(\mathcal{P}, \boldsymbol{q}) \tag{6}$$

Let $r(p)$ be the rank of a given packet $p \in \mathcal{P}$, and let $r_p(p,\ \boldsymbol{q})$ be the rank *perceived* as a result of the mapping decision, which is identified as the highest rank amongst those of packets sharing the same queue. Considering that the objective for the bound vector $\boldsymbol{q}$ is to perfectly approximate PIFO behaviors, we can estimate the unpifoness at enqueue as:

$$\mathcal{U}(\mathcal{P}, \boldsymbol{q}) = \sum_{p \in \mathcal{P}} \text{cost}_{\boldsymbol{q}}(p) \tag{7}$$

where

$$\text{cost}_{\boldsymbol{q}}(p) = r_p(p,\ \boldsymbol{q}) - r(p) \tag{8}$$

Computing the rank perceived requires determining the highest rank among all packets sharing the queue at any given moment. This not only requires to keep track of all ranks in each queue, but also selecting the highest, which is computationally expensive. Since one of the premises of SP-PIFO is to be implementable in the data plane, we relax this condition and keep track of only a single parameter q_i per queue. These parameters, the bounds $\boldsymbol{q}$, simplify the cost estimation of a potential mapping decision at enqueue.

We discuss how we update these parameters as well as the tradeoffs of this relaxation below.

Stage 1: "Push-up" The first stage increases $\boldsymbol{q}$ to minimize the unpifoness of the queue to which the incoming packet is mapped. Specifically, the mapping process scans the queues bottom-up and enqueues the packet in the first queue that satisfies $r(p) \geq q_i$. It then increases q_i to the rank of the enqueued packet. By doing so, the mechanism minimizes (i) the cost for each packet p (at enqueue time); as well as (ii) the impact that this decision may have on future packets.

This mapping process guarantees a zero-cost packet allocation for all packets within a queue. That is, as we effectively keep track of the highest rank per queue, we ensure that no packet with lower rank is mapped to the same queue. This holds for all queues except for the highest-priority queue. There, packets are enqueued even if $r(p) < q_1$.

Algorithm 1 SP-PIFO adaptation algorithm

```
Require: An incoming packet with rank $r$.
 1: procedure PUSH-UP
 2:     for $q_i : q_1$ to $q_n$, $q_i \in \boldsymbol{q}$ do          ▷ Scan bottom-up
 3:         if $r \geq q_i$ or $i = 1$ then
 4:             $q_i \leftarrow r$                        ▷ Update queue bound
 5:             ENQUEUE$(r, i)$                           ▷ Select queue
 6: procedure PUSH-DOWN
 7:     if $r < q_1$ then                                 ▷ Detect inversion
 8:         $cost \leftarrow q_i - r$                     ▷ Compute cost inversion
 9:         for $q_j \in \boldsymbol{q}$, $j \neq i$ do
10:             $q_j \leftarrow q_j - cost$               ▷ Adapt queue bounds
```

Stage 2: "Push-down" As illustrated in §2, the first stage can lead to inversions in the highest-priority queue. The second stage aims at counteracting that effect by reducing the number of ranks enqueued in the highest-priority queue. This is achieved by decreasing *all* queue bounds by some given amount. Different decreasing strategies exist. In SP-PIFO, we decrease each q_i proportionally to the cost of the inversion. That is, we decrease all queue bounds by $q_1 - r(p)$. This choice is both (i) practical, as it can be efficiently implemented in hardware; and (ii) functional, as it results in a reasonable balance between inversions in the highest-priority queue and shifts in the other queues. Below, we provide some insights on the nature of this balance and why it is important for a good PIFO approximation. We simulate the performance of different decreasing strategies in §4.2.

Tradeoffs Unlike the gradient-based algorithm (§3.2), SP-PIFO may converge to a sub-optimal solution exhibiting inversions. One can distinguish three sources of inversions. First, there can be inversions in the highest-priority queue. These inversions are proportional to the probability of observing packets with rank $r(p) < q_1$. Second, after the "push-down" stage, the queue bounds do not necessarily match the highest rank packet in the queue anymore. This may lead to inversions for future packets and is proportional to how often, and how much, queue bounds are decreased. Finally, because only the highest rank in a queue is tracked, it can happen that a packet is enqueued in a higher-priority queue because $r(p) < q_i$, while $r(p)$ is greater than the *lowest* rank in queue i, causing an inversion. This is proportional to the number of ranks between the minimum rank in the queue and the queue bound.

Average-case analysis The exact amount of inversions introduced by each of these three sources is hard to quantify as queue bounds are shifting with (almost) every packet. Yet, *on average*, we can show that the dynamics of SP-PIFO counteract all three sources. On the one hand, it equalizes the probability of $r(p) < q_1$ with the probability of packets being mapped to a specific queue, striking a balance between inversions because there are no higher-priority queues, and inversions because of queue bound mismatch. Furthermore, for this equalizing, the probabilities of specific ranks are weighted more if they are far away from queue bounds, which keeps queues more compact to reduce the chance of overlap.

As a result, on average workloads, SP-PIFO provides a good approximation, and can adapt to arbitrary rank distributions. Nevertheless, there are adversarial packet orderings circumventing these mechanisms, resulting in large unpifoness (§7). We provide the theoretical foundations for these statements in Appendix B and verify them by simulation in §4.2.

4.2 SP-PIFO analysis

We now dive deeper into understanding SP-PIFO using switch-level simulations. We compare its behavior to that of an ideal PIFO queue, along with several well-known scheduling schemes (e.g., FIFO). We first describe the high-level behavior using a uniform rank distribution (§4.2.1), before systematically exploring the design space (§4.2.2).

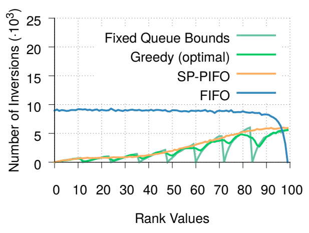
Methodology We implement various scheduling schemes (including SP-PIFO, FIFO, and our gradient-based algorithm) in Netbench [3, 15], a packet-level simulator. We analyze the performance of a single switch scheduling 1500 flows of 1MB (fixed), which start according to a Poisson distribution. We run the simulation during one second. We limit the transmission through an output link of 10 Gbps which corresponds to an average port utilization of 75%. We measure the number of inversions *generated by each rank* at *dequeue*. Whenever a packet is polled, we check whether its rank is higher than any of the ranks remaining at any of the queues. When this occurs, we count an inversion *to the rank generating it* (i.e., the one of the polled packet), making sure that inversions are counted at most once per polled-packet, regardless of the number of packets affected by it.

We compare four scheduling schemes: (i) SP-PIFO (§4); (ii) the gradient-based algorithm (§3, see implementation in A.2); (iii) a strict-priority scheme fixed to the optimal mapping for a uniform distribution (i.e., bounds distributed uniformly across ranks, $q_i = 12i$); and (vi) a FIFO queue, as baseline. All strict-priority schemes (SP schemes) use 8 queues of 10 packets, while the FIFO queue has a capacity of 80 packets.

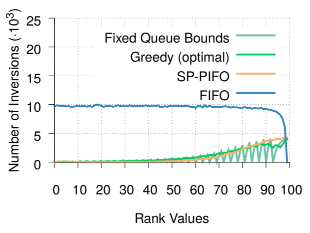
4.2.1 Characterizing general SP-PIFO behavior

We start by showcasing how SP-PIFO handles inversions by analyzing its behavior under a uniform rank distribution. That is, we tag the packets with a rank drawn from a uniform distribution (between 0 to 100).

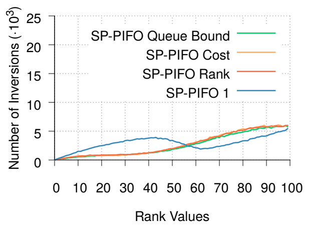
Fig. 5a illustrates the number of inversions generated by each rank for the different SP schemes in comparison with FIFO. We see that a FIFO queue generates a uniform number of inversions across all ranks (since they all share the same queue). In contrast, SP schemes (all the others in Fig. 5a) generate a progressively-higher number of inversions as rank val-



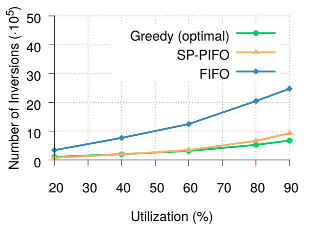
(a) Uniform 8 queues



(b) Uniform 32 queues

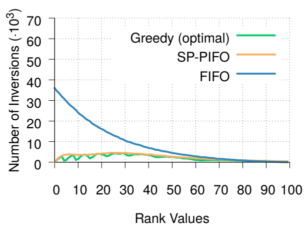


(c) Adaptation strategies

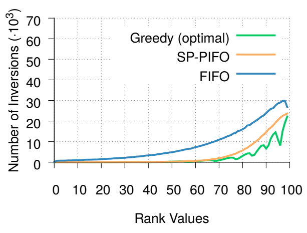


(d) Utilization

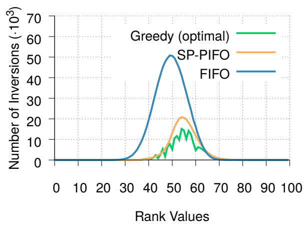
Figure 5: SP-PIFO performance (uniform rank distribution).



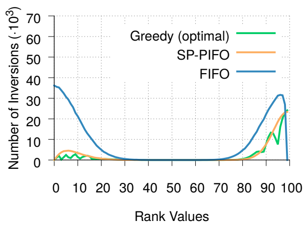
(a) Exponential



(b) Inverse exponential



(c) Poisson



(d) Convex

Figure 6: SP-PIFO performance (alternative distributions).

ues increase. This occurs as higher ranks are mapped to lower-priority queues, which drain packets less frequently. Since those queues have a higher occupancy on average, the potential number of inversions increases. This behavior, however, is not preserved for the lowest-priority queue (the far-right peak in the graph) as a result of starvation. Despite having the largest average queue size, this queue drains fewer packets and, as such, the number of inversions it sees decreases.

For the fixed-queue bounds, we see that a saw-shape delineates the inversions observed across ranks in different queues, reaching the x axis for the ranks corresponding to the queue bounds. Indeed, the lowest rank within each queue never generates inversions since the other ranks sharing the queue have higher values. The second-lowest rank can only generate inversions to the lowest, and the progression continues until the highest rank, which can generate inversions to all the lower ranks sharing the queue.

When considering the gradient-based greedy algorithm (which is optimal) and SP-PIFO, we see that the saw-shape vanishes. This is because queue bounds are not fixed anymore and successive packets of a given rank can be mapped to multiple queues. In particular, since the rank distribution sampled at each adaptation window varies, the queue-bound design in the gradient-based algorithm oscillates. In SP-PIFO, as a higher variability is produced, the number of inversions delineates the *envelope* of the optimal schemes.

4.2.2 Characterizing SP-PIFO design space

We now systematically explore the design space of SP-PIFO along four dimensions: the number of queues, the adaptation strategy when encountering an inversion (in the push-down stage, §4.1), the utilization levels, and the rank distributions.

SP-PIFO manages to approximate the optimal algorithms in all rank distributions and utilization levels, with as little as 8 queues. The best performances are obtained under low utilizations and with 32 queues.

Number of queues (Fig. 5b) When using only 8 queues, SP-PIFO is already within ∼20–29% of the gradient-descent algorithm and the optimal mapping. With 32 queues, it gets even closer, producing only ∼22% more inversions than the optimal and achieving on-par behavior to the gradient-descent algorithm. Overall, it improves FIFO performance ∼3.3× (resp. ∼10×) when only 8 (resp. 32) queues are used.

Push-down strategies (Fig. 5c) We evaluate four adaptation strategies for decreasing each queue bound in the push-down stage: (i) to the value of the next-higher queue bound (“Queue Bound”); (ii) by the cost of the inversion ($q_1 - r(p)$, the strategy in SP-PIFO, “Cost”); (iii) by the rank of the packet causing the inversion (“Rank”); and (iv) by 1 (“1”).

The best performance is obtained for “Queue Bound”, which produces ∼15% more inversions than the gradient-based algorithm. This is followed by “Cost” and “Rank”, with ∼22%, and “1” with ∼33%. While the three first techniques produce similar results, the “push down” effect of “1” is too small to balance the “push up” stage, resulting in many inversions. While “Queue Bound” is marginally better than “Cost”, it is more costly to implement, thus SP-PIFO uses the latter.

Utilization (Fig. 5d) SP-PIFO performance is close to the gradient-based algorithm. For utilizations below 60%, SP-PIFO is on-par with the gradient-based algorithm. The number of inversions slightly increases at higher utilizations: 26% and 38% for 80% and 90%.

Rank distributions (Fig. 6) We analyze the performance of SP-PIFO under four alternative rank distributions: exponential, inverse exponential, Poisson and convex. SP-PIFO performs better than FIFO and is close to the gradient-based algorithm for each distribution.

The performance of SP-PIFO is better for rank distributions in which more ranks appear in higher-priority queues. The number of inversions for SP-PIFO in convex and exponential distributions is only ∼21–24% higher than the gradient-based algorithm. The corresponding numbers for Poisson and inverse exponential amount to ∼49–55%. In all cases, SP-PIFO performs between ∼2.5–3.5× better than a FIFO, with only 8 priority queues.

5 Implementation

In this section, we describe our implementation of SP-PIFO in $P4_{16}$ [7] and $P4_{14}$.[2] Our implementation follows the algorithm described in §4 and spans 190 ($P4_{16}$) and 735 ($P4_{14}$) lines of code. It performs three main operations: (i) computing/extracting the rank from a packet header; (ii) mapping packets to queues (§2); and (iii) updating the queue bounds.

Rank computation We implemented and tested multiple rank computation functions such as LSTF [17], STFQ [23], and FIFO+ [9] in $P4_{16}$. We note that the reduced memory usage in SP-PIFO leaves room to compute ranks directly on the switch. That said, most ranking algorithms can directly be computed by the end-hosts [17].

Mapping We store the queue-bound values in individual registers and access them sequentially using an `if-else` conditional tree. For each register access, we leverage the ALU to perform three operations: (i) we read the queue-bound value and compare it to the packet rank; (ii) we notify the queue-selection result to the control flow using a single-bit metadata; and (iii) we update the queue-bound value to the packet rank if the queue is selected. In the ALU of the last queue, instead of transferring the mapping decision to the control flow using a binary metadata, we first check whether an inversion has occurred before transferring the potential inversion cost using larger metadata.

Adaptation When the mapping process detects an inversion, we need to update all queue bounds. While accessing multiple registers is not restricted by the P4 specification [10], current architectures do not support it (among others, to guarantee line rate). We address this problem by relying on the packet-resubmission primitive to access the queue bounds a second time and update them with the measured inversion cost. While resubmission can possibly break the line-rate guarantees, we only require it occasionally, upon inversions.

[2]The $P4_{14}$ code is used for running SP-PIFO on the Tofino platform [2].

Memory requirements Our implementation only requires n registers where n is the number of queues. We leverage n ALUs to access registers during the mapping process and $n-1$ additional ALUs to update registers from the resubmission pipeline in case of inversions. We use $n-1$ bits of metadata to access the mapping results of non-top-priority queues in their respective ALUs from the control flow (i.e., a single 1-bit metadata field for each queue) and an extra 32-bit field for the top-priority queue to (potentially) transfer the inversion cost.

Regarding the number of stages, our implementation uses more stages than the number of queues in order to perform the sequential access to queue-bound registers during the mapping process. Note that alternative designs would be possible but would come at the expense of line-rate guarantees.

6 Evaluation

We now evaluate SP-PIFO performance and practicality. We first use packet-level simulations to evaluate how SP-PIFO approximates well-known scheduling objectives under realistic traffic workloads (§6.1). We then evaluate SP-PIFO scheduling performance when deployed on hardware switches (§6.2).

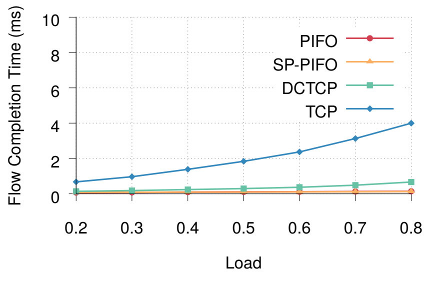
6.1 Performance analysis

We consider two scheduling objectives: (i) minimizing Flow Completion Times (FCTs); and (ii) enforcing fairness. We consider that ranks are set at the end hosts for the former objective and computed in the switch for the latter. For both objectives, we show that SP-PIFO scheduling capabilities achieve near-optimal performance, with as little as 8 queues.

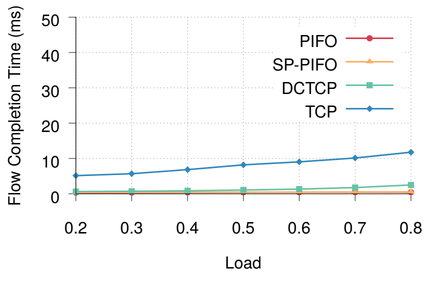
Methodology We integrated SP-PIFO in Netbench [3, 15], a packet-level simulator. Similar to [4], we use a leaf-spine topology with 144 servers connected through 9 leaf and 4 spine switches. We set the access and leaf-spine links to 1Gbps and 4Gbps, respectively. This results in a theoretical end-to-end Round-Trip-Time (RTT) of 32.12μs when crossing the spine (4 hops) and 26μs under the leaf (2 hops). We generate traffic flows following two widely-used heavy-tailed workloads: pFabric web application and data mining [4]. Flow arrivals are Poisson-distributed and we adapt their starting rates to achieve different utilization levels. We use ECMP and draw source-destination pairs uniformly at random.

6.1.1 Minimizing Flow Completion Times

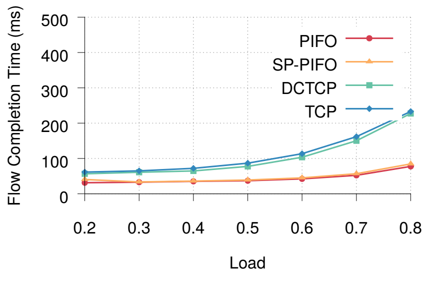
Rank definition & benchmarks We minimize FCTs by implementing the pFabric algorithm [4] which sets the packet ranks according to their remaining flow sizes. Specifically, we compare pFabric performance when run on top of PIFO and SP-PIFO. We also analyze TCP NewReno with traditional drop-tail queues and DCTCP with ECN-marking drop-tail



(a) (0,100KB): Average

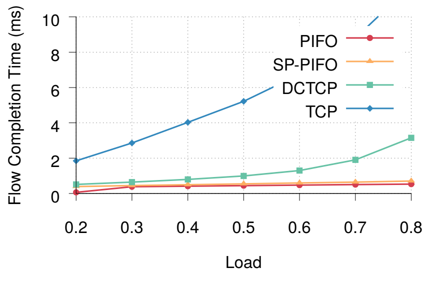


(b) (0,100KB): 99th percentile

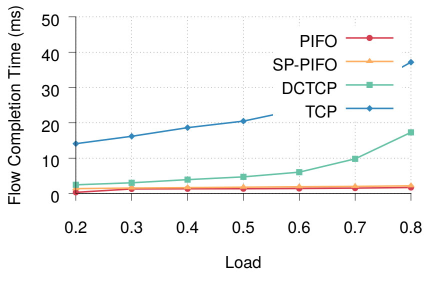


(c) [1MB,∞): Average

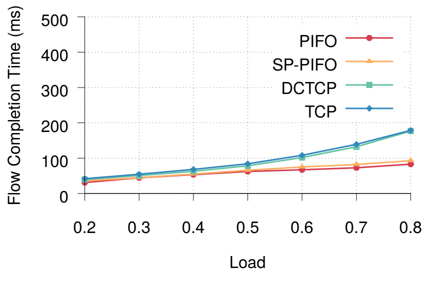
Figure 7: pFabric: FCT statistics across different flow sizes in data mining workload.



(a) (0,100KB): Average



(b) (0,100KB): 99th percentile



(c) [1MB,∞): Average

Figure 8: pFabric: FCT statistics across different flow sizes in web search workload.

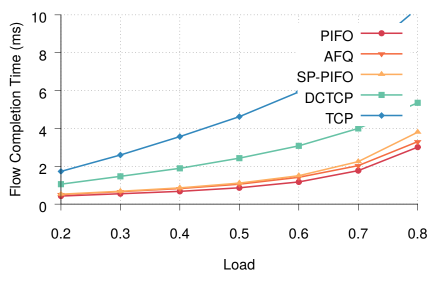
queues. Our pFabric implementation does not consider starvation prevention. As suggested in [4], we approximate pFabric rate control by using standard TCP with a retransmission time-out of 3 RTTs, balancing the difference in RTOs between schemes with the proportional queue size. That is, we use an RTO of 96μs and 8 queues×10 packets for SP-PIFO (resp. 1 queue×80 packets in PIFO), and an RTO of 300μs and 146KB drop-tail queues for both TCP and DCTCP, with ECN marking at 14.6KB, i.e. ∼10 packets.

Summary Fig. 7 and Fig. 8 depict the average and 99th percentile FCTs of large (≥ 1MB) and small flows (< 100KB) for both data mining and web search workloads. We see that SP-PIFO achieves close-to-PIFO performance in both distributions. When comparing performance across flow sizes, we see that SP-PIFO achieves better performance for small flows. This is not surprising since those flows are mapped into higher-priority queues. As discussed in §4.2, strict-priority schemes provide higher unpifoness protection for packets mapped into higher-priority queues.

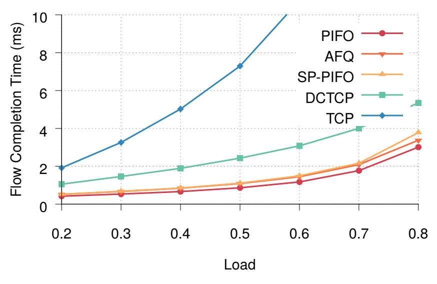
When comparing the two traffic distributions, we see that SP-PIFO performs better under the data mining workload. This is again expected. While both distributions are heavy-tailed, the data mining one is more skewed [4] and therefore easier to handle for SP-PIFO. Indeed, the probability of having large flows simultaneously sharing the same port (potentially blocking smaller flows) is lower for the data mining workload.

Data mining (Fig. 7) The average FCTs achieved by PIFO and SP-PIFO are similar for small flows, i.e. within ∼0.4–11%. Concretely, SP-PIFO outperforms DCTCP and TCP by a factor of ∼2–5× and ∼8–30×, respectively. When considering the 99th percentile, the gap between PIFO and SP-PIFO slightly accentuates to ∼9.6–26.6%. Still, SP-PIFO outperforms DCTCP and TCP by a factor of ∼1.5–4.7× and ∼12.5–22×, respectively. The largest performance gap between PIFO and SP-PIFO occurs at low utilization. In this regime, the number of packets scheduled is low and the transient adaptation of SP-PIFO is more visible. Whenever the utilization is >40%, the difference is consistently below 20%. Finally, SP-PIFO and PIFO still perform similarly among large flows: within ∼1.9–9%, representing improvements with respect to TCP and DCTCP of ∼1.4–2.7× and ∼1.5–2.8×, respectively.

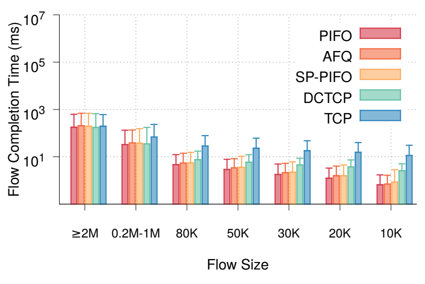
Web search (Fig. 8) The results are similar to the data mining one, with slightly worse performance for SP-PIFO, especially amongst big flows. Indeed, since the distribution is less skewed, bigger flows have higher chances to reach higher-priority queues, blocking transmissions of smaller flows. Still, we see that the performance of SP-PIFO is within ∼16.54–32.5% of PIFO for small flows, and between ∼1.3–4.4× and ∼4.7–16.7× better than DCTCP and TCP. Even at the 99th percentile, the difference between SP-PIFO and PIFO stays within ∼20.7–32%. Note that, while the percentages might seem high, the values we are looking at are very small.



(a) (0,100KB): Average on 8 queues



(b) (0,100KB): Average on 32 queues



(c) FCT breakdown 70%: Average on 32 queues

Figure 9: Fairness: FCT statistics for all flows at different loads, over the web search workload.

6.1.2 Enforcing fairness across flows

Rank definition & benchmarks We enforce fairness across flows by implementing the Start-Time Fair Queueing (STFQ) rank design [13] on top of PIFO and SP-PIFO. We benchmark our solution with AFQ [21] (§8). We analyze the performance for different flow sizes and number of queues. Specifically, we use 8 queues×10 packets in SP-schemes (resp. 1 queue×80 packets for single-queue schemes) and 32 queues×10 packets in SP-schemes (resp. 1 queue×320 packets for single-queue schemes). For AFQ, we select the bytes-per-round parameter which gives the best performance. In our testbed, this is 320 and 80 BpR for the 8-queue and 32-queue scenario, respectively. As in [21], we use DCTCP as transport layer for AFQ, PIFO and SP-PIFO (with an RTO of 300μs). We set ECN marking to 48KB, i.e. ∼32 packets. We generate traffic following the pFabric web search distribution.

Summary Fig. 9a and Fig. 9b depict the average FCTs of small flows across different levels of utilization, when 8 queues and 32 queues are used. Fig. 9c depicts the FCTs across flow sizes at 70% utilization and for 32 queues. In all cases SP-PIFO achieves near-PIFO behavior and is on-par performance with AFQ (current state-of-the-art).

Impact of the utilization (Fig. 9a & Fig. 9b) SP-PIFO stays within ∼23–28% (resp. ∼21–28%) of ideal PIFO across all levels of utilization when 8 queues (resp. 32) are used. Even in the highest utilizations, it is consistently below ∼26% (resp. ∼25%). SP-PIFO performance is at the level of AFQ, within ∼3–10% (resp. ∼0.5–11%), generating improvements of ∼1.4–2.3× and ∼2.7–4.2× (resp. ∼1.4–2.3× and ∼3.7–7.4×) over DCTCP and TCP. The fact that SP-PIFO performance is equivalent with 8 and 32 queues is not surprising: as the bandwidth-delay product is low, only a reduced queue size is required for efficient link utilization.

Impact of flow sizes (Fig. 9c) At 70% utilization, we see that SP-PIFO lies within ∼10–30% of PIFO performance for all flow sizes and is on-par with AFQ. The only exception is for very small flows (<10K) in which AFQ performs 20% better. SP-PIFO improves DCTCP and TCP behaviors for small flows, within ∼1.5–3X and ∼2–13X, respectively. Considering the 99th percentile, we see that SP-PIFO stays within ∼8–35% of PIFO and improves between ∼12–78% and ∼1.5–10.76× with respect to DCTCP and TCP.

Impact of the number of queues (Fig. 10) We analyze the impact of the number of queues on average FCTs for both AFQ and SP-PIFO. We set the BpR at MSS for all queue configurations, as in [21], avoiding AFQ dropping packets too often for cases of fewer queues. We see that while AFQ has a higher sensitivity with respect to the number of queues, SP-PIFO preserves a similar level of performance, without any configuration or prior traffic knowledge.

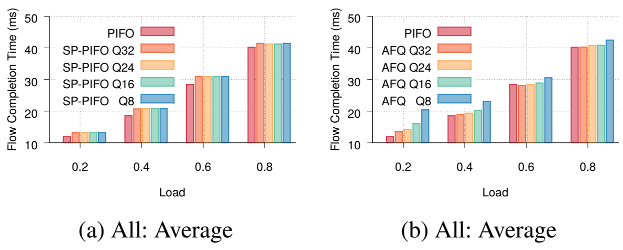
6.2 Hardware testbed

We finally evaluate our hardware-based implementation of SP-PIFO on the Barefoot Tofino Wedge 100BF-32X platform [2]. We perform two experiments. First, we analyze the bandwidth allocated by SP-PIFO to flows with different ranks when scheduled over a bottleneck link. Second, we measure the impact on the FCT when SP-PIFO runs pFabric. We show that SP-PIFO efficiently schedules traffic at potentially Tbps.

Bandwidth shares We transmit 8 UDP flows of 20Gbps between two servers. We generate the flows progressively, in increasing order of priority (decreasing rank). We use 4 priority queues and schedule the flows over a 10Gbps interface. We generate the flows using Moongen [12] and use an intermediate switch to amplify them to the required throughput.

Fig. 11 depicts the flows' bandwidth and how SP-PIFO manages to virtually extend the number of queues. As expected, the first 4 flows receive the complete bandwidth, since they are mapped to dedicated queues. As the number of flows exceeds the number of queues, flows start to share queue space and see a reduced bandwidth.

Flow completion times We simultaneously generate 1000 TCP flows of different sizes, going from 1GB to 100GB in steps of 100MB, and schedule them over a bottleneck link of 7Gbps. We set the rank of each flow to the absolute flow size,



(a) All: Average

(b) All: Average

Figure 10: Fairness: FCT statistics for all flows at different loads, when the number of queues is modified.

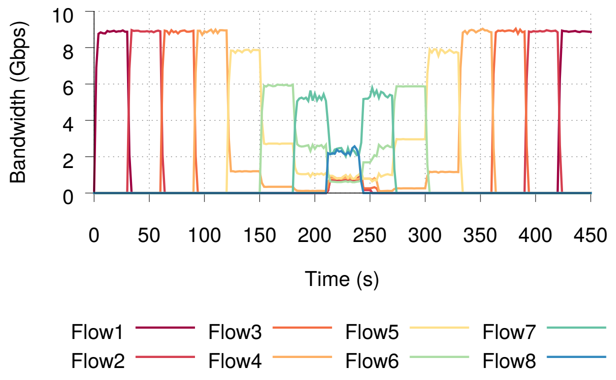


Figure 11: Tofino: Bandwidth allocation under progressive flow generation with increasing priorities.

following [4]. We compare the FCTs achieved by SP-PIFO scheduling and the ones achieved by a FIFO queue.

Fig. 12 shows the resulting FCTs. As expected, the FIFO queue leads to increased FCTs by not considering flow size. In contrast, SP-PIFO prioritizes short flows over long ones, minimizing their FCTs and the overall transmission time.

7 Discussion

In this section, we discuss the limitations of SP-PIFO and how we can mitigate them. We first discuss intrinsic limitations that come from using PIFO as a scheduling scheme. We then discuss specific limitations of SP-PIFO together with the problem of adversarial workloads. Finally, we suggest potential hardware primitives that could facilitate PIFO implementations in the future.

PIFO-inherited limitations Individual PIFO queues suffer from two main limitations. First, they cannot rate-limit their egress throughput preventing them from implementing non-work-conserving scheduling algorithms. SP-PIFO also shares the same limitation. Second, PIFO queues cannot directly implement hierarchical scheduling algorithms. Yet, as proposed by [23], multiple SP-PIFO schemes (i.e., using different set of priority queues) can be grouped as a tree to approximate hierarchical scheduling algorithms. The key challenge consists

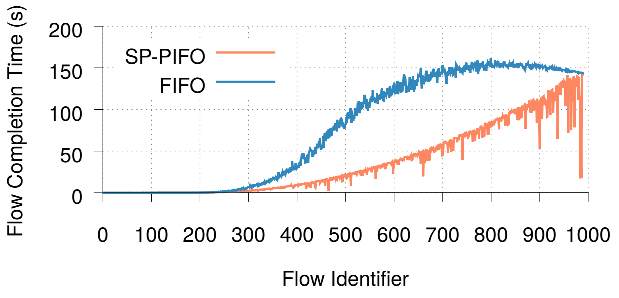


Figure 12: Tofino: FCT statistics across different flow sizes with pFabric ranks.

in figuring out how to allow access of multiple queues with existing traffic manager capabilities. While this is orthogonal to this paper, one option would be to recirculate packets, enabling access to the traffic manager (and therefore the queues) multiple times in the same pipeline. Doing so, while limiting the impact on performance, is an interesting open question.

SP-PIFO-specific limitations The main limitation of SP-PIFO is that, as an approximation scheme, it cannot guarantee to perfectly emulate the behavior of a theoretical PIFO queue for all ranks. We note two things. First, our evaluation (§6) shows that, for realistic workloads, SP-PIFO performance is often on-par with PIFO performances. Second, we note that SP-PIFO *can* provide strong PIFO-like guarantees *for some ranks* by dedicating some queues to them at the price of reduced performance for the other ranks. We believe this is an interesting tradeoff as current switches can support up to 32 queues per port [21].

Adversarial workloads We have argued that, on average, SP-PIFO can adapt to any kind of rank distribution. This has certain limitations. First, we assume that all queues are drained at some point. Nonetheless, a malicious host could send a large number of high-priority packets and, as a result, packets in lower-priority queues would never be drained. Such "starvation" attacks are common to any type of priority scheme. For instance, a malicious host could try to grab a bigger slice of the network resources by setting ranks to 0 in slack-based algorithms [4,9,17] or resetting flow identifiers in fair-queuing schemes [23]. The problem of starvation in strict-priority scheduling is also well-known in the context of QoS and is typically addressed by policing high-priority traffic at the edge of the network [18].

Aside from starvation attacks we also assume that, for a given rank distribution, the particular order of ranks is random. In practice, this is reasonable. While the ranks for individual flows might have some structure (e.g., monotonically-increasing ranks), when various flows are scheduled together the ordering of their packet ranks is randomized. Yet, attackers could try to coordinate large numbers of flows to create adversarial orderings, which "outplay" the adaptation mechanisms (§B.3). Nevertheless, any non-malicious flow which is active at the same time can thwart such strategies by randomly breaking the adversarial order. Aside from that, the network could be monitored to detect such adversarial attacks.

Facilitating PIFO in the future On a forward-looking perspective, we note some improvements in hardware primitives that would facilitate PIFO implementations in the future. As we already discussed in §5, a higher number of stages would facilitate per-queue state storage and a higher number of queues would directly increase PIFO performance. Further than that, multiple and dynamic memory access between the ingress and egress pipelines would allow state updates after inversions in the highest-priority queue without having to rely on resubmission techniques. In the same direction, access to queue information from the ingress pipeline or an enhanced flexibility in the management of strict-priority queues directly from the data plane would enable more accurate unpifoness prediction at enqueue, opening the doors to higher-performance SP-PIFO algorithms.

8 Related work

Programmable packet scheduling While scheduling has been extensively studied over the years, the idea of making it programmable is relatively recent [17,22]. In [24], Sivaraman et. al. suggested programmable scheduling by proving that the best scheduling algorithm to use depends on the desired performance objective. In [17], Mittal et. al. made the observation that, despite certain algorithms accept configurations to approximate a wide range of objectives, a universal packet scheduling outperforming in all scenarios does not exist.

Several abstractions for programmable scheduling have been proposed afterwards. In addition to PIFO [24], Eiffel [19] presents an alternative queue structure which approximates fine priorities by exploiting the characteristics that define packet ranks in most scenarios to diminish the required computational complexity. In contrast to [19, 24], which rely on new hardware designs, SP-PIFO shows that efficient programmable packet scheduling can be achieved today, at scale, and on existing devices.

Exploiting priority queues Other (recent) schemes leverage multiple priority queues for specific performance objectives. They highlight the need of programmable scheduling in existing devices [16], and illustrate how rank designs producing close-to-optimal results can already be implemented in existing data planes. For enforcing fairness, FDPA [8] simplifies the computational cost of per-flow virtual counters or individual user queues in traditional-fair-queuing schemes by using arrival-rate information at a user level. AFQ [21], instead, emulates ideal fair queuing by implementing per-flow counters on a count-min sketch and dynamically rotating priorities in a strict-priority scheme to imitate the round-robin behavior. SP-PIFO differs by fixing queue priorities and dynamically adapting the mapping of packets to those queues. This actually makes SP-PIFO implementable *at line rate* in existing data planes.

pFabric [4] and PIAS [5] show the use of priority queues in flow completion time minimization. While pFabric relies in general on a PIFO-queue design, [4] includes experiments in which flows are mapped to priority queues based on their size. While pFabric experiments use thresholds fixed from the knowledge of flow distributions, SP-PIFO adapts the mapping design automatically per-packet, without any traffic knowledge required in advance. PIAS [5] approaches the case of unknown flow sizes and uses Multi-level Feedback Queues (MLFQ) [11] to achieve the desired Shortest Job First (SJF) behavior, by gradually switching flows from higher to lower-priority queues as their number of transmitted bytes increase.

In contrast to these proposals, SP-PIFO supports a much wider range of performance objectives. SP-PIFO (like PIFO [24]) can be used to implement *any* scheduling algorithm in which the relative scheduling order does not change with future packet arrivals. As illustrated in the evaluation section (§6), the algorithms presented in AFQ [21], FDPA [8], pFabric [4] and PIAS [5] can be used as ranking designs (i.e., setting packet ranks to scheduling virtual rounds, estimated arrival rates, shortest remaining processing time of flows, or number of packets transmitted under the MLFQ aging design) to be run on top of SP-PIFO.

9 Conclusions

We presented SP-PIFO, a programmable packet scheduler which closely approximates the theoretical behavior of PIFO queues, today, on programmable data planes. The key insight behind SP-PIFO is to dynamically adapt the mapping between the packet ranks and a (fixed) set of strict-priority queues.

Our evaluation on realistic workloads shows that SP-PIFO is practical: it closely approximates PIFO behaviors and, in many cases, perfectly matches them. We also confirm that SP-PIFO runs on actual programmable hardware.

Overall, we believe that our work shows that the benefits of programmable packet scheduling—experimenting with new scheduling algorithms—can be fulfilled today, in existing networks.

Acknowledgments

We are grateful to the NSDI reviewers and our shepherd, Anirudh Sivaraman, for their insightful comments. We also thank the members of the Networked Systems Group at ETH Zürich (especially Edgar Costa Molero) together with Changhoon Kim from Barefoot for their valuable feedback.

References

[1] Broadcom Trident II. https://www.broadcom.com/products/Switching/DataCenter/BCM56850-Series, 2016.

[2] Barefoot Tofino. http://barefootnetworks.com/products/brief-tofino/, 2017.

[3] Netbench. http://github.com/ndal-eth/netbench, 2018.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *ACM SIGCOMM*, Hong Kong, China, 2013.

[5] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[6] Alexander Barkalov, Larysa Titarenko, and Malgorzata Mazurkiewicz. *Foundations of Embedded Systems*. Springer International Publishing, 2019.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. 2014.

[8] Carmelo Cascone, Nicola Bonelli, Luca Bianchi, Antonio Capone, and Brunilde Sansò. Towards Approximate Fair Bandwidth Sharing via Dynamic Priority Queuing. In *IEEE LANMAN*, Osaka, Japan, 2017.

[9] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *ACM SIGCOMM*, Baltimore, MD, USA, 1992.

[10] The P4 Language Consortium. P4-16 Language Specification, version 1.1.0-rc. https://p4.org/p4-spec/docs/P4-16-v1.1.0-draft.pdf, 2018.

[11] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. An Experimental Time-sharing System. In *ACM AIEE-IRE*, New York, NY, USA, 1962.

[12] Sebastian Gallenmüller, Paul Emmerich, Daniel Raumer, and Georg Carle. MoonGen: Software Packet Generation for 10 Gbit and Beyond. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[13] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *ACM SIGCOMM*, Palo Alto, CA, USA, 1996.

[14] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP*, Shanghai, China, 2017.

[15] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. Beyond Fat-trees Without Antennae, Mirrors, and Disco-balls. In *ACM SIGCOMM*, Los Angeles, CA, USA, 2017.

[16] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on Load Distribution and the Role of Programmable Switches. In *ACM SIGCOMM*, New York, NY, USA, 2019.

[17] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal Packet Scheduling. In *USENIX NSDI*, Santa Clara, CA, USA, 2016.

[18] Juniper Networks. Class of Service Feature Guide for Security Devices. page 115, 2018.

[19] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. Eiffel: Efficient and Flexible Software Packet Scheduling. In *USENIX NSDI*, Boston, MA, USA, 2019.

[20] Chuck Semeria. Supporting Differentiated Service Classes: Queue Scheduling Disciplines. In *Juniper Networks White Paper*, Sunnyvale, CA, USA, 2001.

[21] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *USENIX NSDI*, Renton, WA, USA, 2018.

[22] Anirudh Sivaraman, Suvinay Subramanian, Anurag Agrawal, Sharad Chole, Shang-Tse Chuang, Tom Edsall, Mohammad Alizadeh, Sachin Katti, Nick McKeown, and Hari Balakrishnan. Towards Programmable Packet Scheduling. In *ACM HotNets*, Philadelphia, PA, USA, 2015.

[23] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *ACM SIGCOMM*, Florianopolis, Brazil, 2016.

[24] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *ACM HotNets*, College Park, MD, USA, 2013.

A Gradient-based algorithm

In this appendix we detail the greedy iterative algorithm presented in §3.2. We first motivate and proof how the algorithm converges to the optimal solution (A.1). Second, we show how to effectively prune the search space making computation efficient while keeping convergence (A.2). Finally, we analyze its implementation (A.3) and convergence requirements (A.4).

A.1 Greedy optimization

The algorithm (alg. 2) iteratively minimizes the risk by adjusting queue bounds, one queue and one step at a time, until reaching convergence. At each iteration, the algorithm predicts, for every q_i, whether moving the bound by one (in either direction) decreases the expected risk, and moves the bound in the direction of maximum decrease. In the following, we discuss first, how the algorithm can predict the expected change in risk, and second, why checking a single step is sufficient to converge.

Algorithm 2 Greedy optimization

```
Require: $k$: Step size, $\boldsymbol{q}_{\text{init}}$: Initial bounds
 1: procedure ADAPTATION
 2:     $\mathcal{D} \leftarrow \emptyset$
 3:     $\boldsymbol{q} \leftarrow \boldsymbol{q}_{\text{init}}$                       ▷ Initialize bounds
 4:     for all $p$: incoming packet do
 5:         $\mathcal{D} \leftarrow \mathcal{D} \cup \{rank(p)\}$             ▷ Collect samples
 6:         if $|\mathcal{D}| = k$ then                        ▷ Adapt bounds
 7:             $\mathcal{P} \leftarrow$ COMPUTERANKPROBABILITES($\mathcal{D}$)
 8:             repeat
 9:                 $\boldsymbol{q} \leftarrow$ UPDATEMAPPING($\boldsymbol{q}$, $\mathcal{P}$)
10:             until $\boldsymbol{q}$ converges
11:             $\mathcal{D} \leftarrow \emptyset$                          ▷ Reset samples
12: function UPDATEMAPPING($\boldsymbol{q}$, $\mathcal{P}$)
13:     for $q_i \in \boldsymbol{q}$ do
14:         $\Delta^+ \leftarrow$ RISKFROMINCREMENT($q_i$, $\mathcal{P}$)
15:         $\Delta^- \leftarrow$ RISKFROMDECREMENT($q_i$, $\mathcal{P}$)
16:         if $(\Delta^+ \leq 0)$ and $(\Delta^+ \leq \Delta^-)$ then
17:             $q_i \leftarrow q_i + 1$
18:         else if $(\Delta^- \leq 0)$ and $(\Delta^- < \Delta^+)$ then
19:             $q_i \leftarrow q_i - 1$
        return $\boldsymbol{q}$
```

Risk difference In §3.2, we demonstrated that the risk can be analyzed on a per-queue basis from the cost of mapping packets with different ranks to the same queue. Consequently, changes in the risk resulting from changing the bound vector $\boldsymbol{q}$ can be analyzed by comparing the risk difference in affected queues. To be precise, every change of a single element q_i in $\boldsymbol{q}$ affects two queues, queue i and $i-1$, as ranks are either moved from i to $i-1$ (increase in q_i) or moved from $i-1$ to i (decrease in q_i).

Theorem 1 *Let $r^* = q_i$, let Q_i be the set of ranks mapped to queue i (before any changes). Increasing q_i by 1 changes the risk by:*

$$\Delta_i^+ = p(r^*)\Big(\sum_{r\in Q_{i-1}} p(r)cost(r^*,r) - \sum_{r\in Q_i} p(r)cost(r,r^*)\Big) \tag{9}$$

Let $r^* = q_i - 1$. Decreasing q_i by 1 changes the risk by:

$$\Delta_i^- = p(r^*)\Big(\sum_{r\in Q_i} p(r)cost(r^*,r) - \sum_{r\in Q_{i-1}} p(r)cost(r,r^*)\Big) \tag{10}$$

Proof Increasing q_i effectively removes the lowest rank from queue i, which now becomes the highest rank in queue $i-1$. As the new highest rank in queue $i-1$, it causes possible inversions and therefore risk for *all* other ranks in queue $i-1$, resulting in the first, positive term in eq. 9. Conversely, as the lowest rank in queue, it was prone to receive inversions from any other element in the queue, supposing a risk in queue i that is removed with the change. This risk reduction results in the second, negative, term.

The proof for decreasing q_i is symmetrical, with the main difference that now, rank q_{i-1} is the one changing from queue $i-1$ to queue i.

Greedy step Based on the theory presented, the algorithm computes the risk and either (for every q_i):

(a) Does not move q_i, if neither incrementing or decrementing reduces the expected risk.

(b) Increments q_i, if incrementing decreases the risk more than decrementing.

(c) Decrements q_i, if decrementing decreases the risk more than incrementing.

This effectively prunes the search space. At every iteration, the algorithm only requires a constant amount of comparisons, and it does not explore directions further in case they increase the risk. In the following, we show why deciding not to explore a direction further after a single step is reasonable.

Theorem 2 *Let Δ_i^+ and Δ_i^- denote the prospective in- and decreases from incrementing/decrementing q_i by 1. Let Δ_i^{++} and Δ_i^{--} denote the in- and decreases from incrementing/decrementing q_i by more than 1. Let the cost function used to compute the differences be non-decreasing in $|r^* - r|$ and 0 if and only if $r^* = r$. Then:*

1. *If $\Delta_i^+ > 0$, then $\Delta_i^{++} > 0$.*

2. *If $\Delta_i^- > 0$, then $\Delta_i^{--} > 0$.*

Proof

1: If $\Delta_i^+ > 0$,

$$\sum_{r \in Q_{i-1}} p(r)\text{cost}(r^*, r) > \sum_{r \in Q_i} p(r)\text{cost}(r, r^*) \tag{11}$$

Let $r^{**} = q_i + 1$, i.e. the second-lowest rank in queue i, which would be moved if we move the queue bound by more than 1. Moving both r^* and r^{**} would cause the following change in risk:

$$\Delta_i^{++} = \tag{12}$$

$$p(r^*)(\sum_{r \in Q_{i-1}} p(r)\text{cost}(r^*, r) - \sum_{r \in Q_i} p(r)\text{cost}(r, r^*)) + \tag{13}$$

$$p(r^{**})(\sum_{r \in Q_{i-1}} p(r)\text{cost}(r^{**}, r) - \sum_{r \in Q_i} p(r)\text{cost}(r, r^{**})) \tag{14}$$

Note that we can omit the cost between r^* and r^{**} in eq. 14: as the cost function is by definition symmetric, the additional increase in the left-hand term is exactly equal in magnitude to the additional decrease in the right-hand term, and thus they cancel each other. Thus we omit the term to not clutter the notation. Next, again by definition of the cost function, if $r^{**} > r^* > r$, then $cost(r^{**}, r) \geq cost(r^*, r)$, and if $r > r^{**} > r^*$, then $cost(r, r^{**}) \leq cost(r, r^*)$. Additionally, we note that the order of arguments in the cost function does not matter, as it is symmetrical. Applied to the risk of the lower- and higher-priority queue respectively (eq. 14), this gives:

$$\begin{gathered}\sum_{r \in Q_{i-1}} p(r)cost(r^{**}, r) \geq \sum_{r \in Q_{i-1}} p(r)cost(r^*, r) \\ \sum_{r \in Q_i} p(r)cost(r, r^{**}) \leq \sum_{r \in Q_i} p(r)cost(r, r^*)\end{gathered} \tag{15}$$

And in conclusion, the left hand term in eq. 14 is larger than the left hand term in eq. 13, and the right hand term in eq. 14 is smaller then the left hand term in eq. 13. Consequently, if eq. 13 is positive, eq. 14 must also be positive (as probabilities are always positive), proving that if one step does increase the risks, two steps will also increase the risk. The exact same procedure can be repeated for larger step sizes, which we omit here.

2: This proof is conceptually identical to the other direction, and we will thus omit it. The guiding principle is the same: moving more than one rank can only cause higher increase in risk in the queue the ranks are moved to, and lower decrease in risk in the queue the ranks are taken from, compared to the previous ranks. Thus, if already moving one rank causes a higher increase in risk in one queue than decrease in the other, moving additional ranks does not change this.

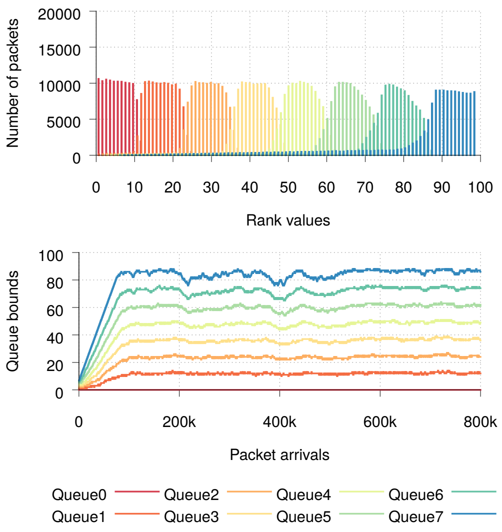


Figure 13: Greedy convergence for uniform rank distribution.

Conclusion We have explained how the greedy algorithm only requires exploring the direction which offers a potential decrease in risk, and we have proved how the risk does not decrease with the distance between ranks (it cannot be better to have a bigger inversion, only equal or worse). This allows the greedy algorithm to quickly decide if a direction is not worth investigating, effectively pruning the search space.

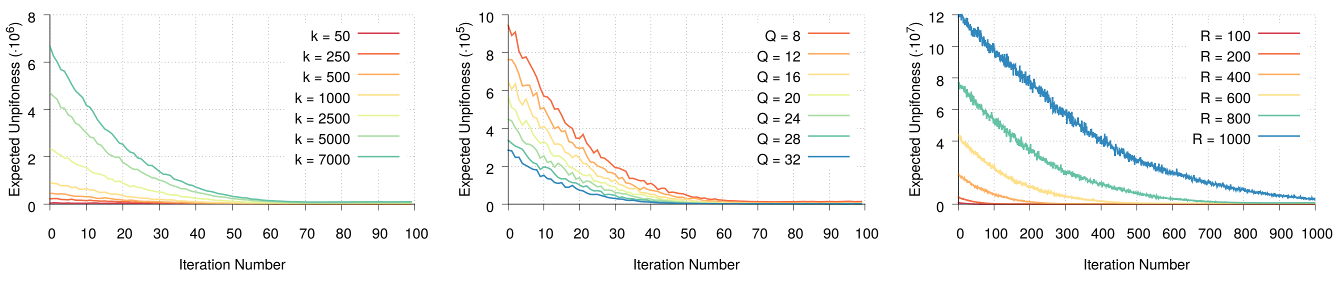
A.2 Efficient computation

As tracking the complete rank distribution at each iteration might be too expensive in terms of memory, and repeating the adaptation until convergence too costly in terms of complexity, we show in the following lines how the mathematical formulation of the problem allows a simplified implementation which only requires 4 counters per queue.

From the empirical probability definition, $p_{\mathcal{D}}(r) = |r_{\mathcal{D}}|/|\mathcal{D}|$, we can rewrite eq. 9 and eq. 10 as:

$$\begin{aligned}\Delta_i^+ &= \frac{|q_i|}{|\mathcal{D}|^2} \cdot (\sum_{r \in Q_{i-1}} |r|\text{cost}(q_i, r) - \sum_{r \in Q_i} |r|\text{cost}(r, q_i)) \\ \Delta_i^- &= \frac{|q_i - 1|}{|\mathcal{D}|^2} \cdot (\sum_{r \in Q_i} |r|\text{cost}(q_i - 1, r) - \sum_{r \in Q_{i-1}} |r|\text{cost}(r, q_i - 1))\end{aligned} \tag{16}$$

Since the queue bound q_i stays constant throughout the adaptation window, each of the summations in eq. 16 can be implemented through a counter which gets updated every time a new packet arrives, with its carried rank. Note that the number of counters required increases linearly with the number of queues. Also, observe that the counters in eq. 16, only allow the computation of one step in the gradient. However, this is enough since, as can be seen in Fig. 13, the one-step version manages to converge in practice.



(a) Convergence vs. adaptation window (b) Convergence vs. number of queues (c) Convergence vs. rank range

Figure 14: Greedy algorithm adaptation microbenchmark.

A.3 Implementation requirements

With the computation presented in A.2, implementing the gradient-based algorithm on top of n priority queues, requires n registers for queue-bound storage and $(4 \cdot n)$ registers for the gradient computation. The mapping process §2 requires packets to potentially read all the queue-bound values (i.e., for packets scheduled in the highest-priority queue). In the same direction, while most packets only need to update the two counters corresponding to their queue, the k_{th} packet in each sequence needs to access *all* counters to perform the adaptation decision. This supposes being able to read $n + (4 \cdot n)$ different registers for a single packet (without even considering the updates). Since existing devices only support up to 12-16 stages, with a single register access per stage [14], the implementation of the greedy algorithm is not feasible for a practical number of queues (i.e., $n \geq 8$).

A.4 Convergence analysis

We now show how the greedy-algorithm performance varies when modifying the three main degrees of freedom: (i) the adaptation window (i.e., the number of packets that are monitored before the adaptation mechanism is executed); (ii) the number of queues available in the strict-priority scheme; and (iii) the number of ranks in the distribution. For that, we analyze the unpifoness evolution of a single switch running the greedy algorithm for a uniform rank distribution from 0 to 100 until convergence. We compute unpifoness as specified in §3.1, based on the packets scheduled and the queue bounds used during the adaptation window.

Effects of varying the adaptation window Fig. 14a shows the unpifoness evolution when we run the greedy algorithm on top of a strict-priority scheme of 8 queues, and we vary the adaptation window from 50 to 7000 packets. We observe that, for the algorithm to converge, the adaptation window needs to be broad enough to cover a *complete* sample of the rank distribution (i.e., one that characterizes all its representative behaviors). In our case, any adaptation window below 100 packets can not characterize completely the rank distribution. Indeed, Fig. 14a depicts how the greedy algorithm correctly converges as soon as more than 200 packets are monitored per iteration. In general, the broader the adaptation window, the more precise the rank distribution estimate, and the better the adaptation decision. However, while a too narrow adaptation window can suppose missing important information of the rank distribution and breaking convergence guarantees, a too broad adaptation window can make the algorithm too slow to converge, negatively impacting the performance.

Finally, the greedy algorithm only converges if the rank distribution has a smaller variability than the adaptation rate (i.e., the rank distribution is stable during the time it takes for the algorithm to converge). Relating it to the previous point, simpler rank distributions, which require narrower adaptation windows, can afford higher levels of variability. In contrast, complex distributions which take longer to adapt and are required to keep stable longer for the algorithm to converge.

Effects of varying the number of queues Fig. 14b depicts the case in which we fix an adaptation window of 1000 packets, and modify the number of queues from 8 to 32. All queues have a constant size of 10 packets. We see how the higher number of queues the lower the unpifoness, and the better the PIFO approximation. This is expected since each queue can be perceived as an opportunity to sort packets with different ranks, and therefore to reduce the number of inversions. Also, we can see how the number of iterations required by the algorithm to converge does not directly depend on the number of queues. This results from the fact that each adaptation decision analyzes (and, if required, updates) potential redesigns for *all* the different queue bounds.

Effects of varying the number of ranks Fig. 14c presents the effects of modifying the range of the uniform rank distribution from 100 to 1000 ranks, when we fix the number of queues to 8 and the adaptation window to 1000 packets. As expected, under the same number of queues, a higher number of ranks implies an increase in unpifoness. Also, as the rank ranges get closer to the adaptation window, the distribution estimates get worse, and the adaptation gets tougher.

B Theoretical analysis of SP-PIFO

SP-PIFO is a highly-dynamic probabilistic system. In particular, its queue bounds $\boldsymbol{q}$ change with nearly every incoming packet. Nevertheless, in this section we show that the system has an attractive equilibrium $\boldsymbol{q}^*$ (B.1), how this equilibrium balances the different causes of inversions (B.2), and we discuss the limitations and open question of our analysis (B.3).

B.1 Stable equilibrium

Queue-bound dynamics Consider SP-PIFO as a discrete-time system, where each time step corresponds to an arriving packet. Let $\boldsymbol{q}^t$ be the queue bounds at step t, when the t-th packet arrives. Then, the queue bounds at step $t+1$ are:

$$\boldsymbol{q}^{t+1} = \boldsymbol{q}^t + \Delta(r^t) \tag{17}$$

where r^t is the rank of the t-th packet, and $\Delta(r^t)$ is the change this packet causes on the queue bounds. The queue-bound change is given by the "push-down" and "push-up" stages of SP-PIFO, respectively. If the packet causes an inversion in the highest-priority queue, all queue bounds are *decreased* by $q_1^t - r^t$. Otherwise, there is exactly one queue i such that $q_i^t \leq r^t < q_{i+1}^t$, and only q_i is set to r^t, or equivalently, is *increased* by $r^t - q_i^t$. Finally, let $p(r^t)$ be the probability of rank r for the t-th packet. Then, the expected value of the queue bounds at step $t+1$, and the expected difference to the queue bounds at step t are, respectively: [3]

$$\mathrm{E}\left[q_i^{t+1}\right] = \mathrm{E}\left[q_i^t\right] \tag{18}$$
$$+ \underbrace{\sum_{q_i^t \leq r^t < q_{i+1}^t} p(r^t)(r^t - q_i^t)}_{\Delta_i^+(\boldsymbol{q}^t, r^t)} \tag{19}$$
$$- \underbrace{\sum_{r^t < q_1^t} p(r^t)(q_1^t - r^t)}_{\Delta^-(\boldsymbol{q}^t, r^t)} \tag{20}$$
$$\Leftrightarrow \mathrm{E}\left[q_i^{t+1} - q_i^t\right] = \Delta_i^+(\boldsymbol{q}^t, r^t) - \Delta^-(\boldsymbol{q}^t, r^t) \tag{21}$$

Equilibrium As expected, we can see from eq. 21 that the change of queue bounds is determined by the "push-up" (Δ_i^+) and "push-down" (Δ^-) stages working against each other. Indeed, if Δ_i^+ is larger than Δ^-, the queue bound increases, and vice versa. The system has an equilibrium $\boldsymbol{q}^*$, where $\Delta_i^+ = \Delta^-$ and the expected change is 0. Note that this equilibrium depends on the rank probability.

Attraction The equilibrium $\boldsymbol{q}^*$ is attractive, i.e. if $q_i^t < q_i^*$, $\mathrm{E}[q_i^{t+1} - q_i^t] > 0$, and vice versa. For small perturbations, this is straightforward. Assume that all queue bounds are in equilibrium, except q_i. If $q_i^t < q_i^*$, then $\Delta_i^+(\boldsymbol{q}^t, r^t) > \Delta_i^+(\boldsymbol{q}^*, r^t)$, because the sum in eq. 19 has (i) more (non-negative) terms; and (ii) each term is weighted stronger, as the difference $r^t - q_i^t$ is larger. On the other hand, $\Delta^-(\boldsymbol{q}^t, r^t)$ is either equal to $\Delta^-(\boldsymbol{q}^*, r^t)$ (for $i > 1$) or even smaller (for $i = 1$, as there are less, and lesser weighted, terms in the sum 20). Thus, the increase is larger than the decrease, and the expected change to q_i is positive. The argument for $q_i^t > q_i^*$ is symmetrical.

For larger disturbances, the equilibrium is also attractive, but it might take more than a single time step, as the "push-up" stage for q_i also depends on q_{i+1}: if both $q_i < q_i^*$ and $q_{i+1} < q_{i+1}^*$, the "push-up" might be too weak to pull q_i towards the equilibrium. However, this is not the case for the lowest-priority queue q_n, for which the "push-up" does not depend on another queue. Thus, lower-priority queues (at least q_n) might be pulled towards the equilibrium at first, while other q_i are not. Notice that an expected increase of q_{i+1}^t increases the "push-up" mechanism for q_i^{t+1} and decreases it for q_{i+1}^{t+1} (eq. 19). Eventually, as the lower-priority queue bound is getting closer to the equilibrium, the higher-priority queue bound is also pulled towards the equilibrium. This continues until the highest-priority queue, where an expected increase of q_1^t also increases the "push-down" mechanism for all bounds at step $t+1$ (eq. 20). As a result, over multiple time steps, the expected effects of the "push-up" and "push-down" stages equalize, eventually pulling all q_i towards q_i^*.

B.2 Balance

As explained in §4, there are three main reasons for unpifoness: (i) inversions in the highest-priority queue, after which all queue bounds are decreased; (ii) inversions in a lower-priority queue after its queue bound has been decreased; (iii) inversions in a lower-priority queue, if its highest rank "overtakes" the lowest rank of a higher-priority queue.

As we can see in eq. 19, eq. 20, and eq. 21, all these factors play a role in the dynamics of SP-PIFO. At the equilibrium, the probability of "push-down", which is exactly the probability of an inversion in the highest-priority queue (weighted by its severity), is equalized with the probability of a packet being mapped to any other queue (again weighted, more on this below). While this does not directly correspond to inversions, the more packets are mapped to lower-priority queues, the higher is the probability of an inversion in those queues after a "push-down". SP-PIFO thus keeps a balance between inversions (i) and (ii), as decreasing (i) would require a stronger "push-down", which would then increase (ii), and vice versa.

Finally, as mentioned above, the ranks in a queue are weighted by how far they are away from the queue bound ($r^t - q_i^t$). This penalizes long (in terms of distinct ranks) queues, which helps to reduce (iii), as the probability for one queue "overtaking" another increases the further the actual queue bound is from the highest-rank packet in the queue, which increases with the length of the queue.

[3] For queue $i = n$, there is no q_{i+1}^t and there is no upper bound on r^t.

B.3 Assumptions and limitations

The analysis presented above is based on a few assumptions, which we argue are justified, yet pose some open questions.

First, we assume that there exists a finite distribution of ranks. This is given in practice. Since ranks need to be processed and stored in hardware, which offers restricted resources, rank ranges must have a limited size.

Second, although SP-PIFO can rapidly adapt to varying rank distributions (in particular faster than the greedy algorithm), we assume that the rank distribution is stable enough such that an equilibrium can exist at all. However, it remains an open question whether there is a point in which the rank-distribution variation might be too fast for the system to actually converge to an equilibrium. In that (hypothetical) case, the analysis presented herein would not be useful to provide any additional insights on the performance of SP-PIFO.

Finally, we assume that the ranks appear in random order, independently from each other. At the first glance, this may seem irrational, as many scheduling algorithms have some structure in the way how ranks are assigned to packets for a given flow. Nevertheless, in practical scenarios, many flows are scheduled together, and even though the ranks for individual flows might be structured, the combined ranks of packets across flows become randomized.

Adversarial workloads Based on the previous assumptions, we have shown that SP-PIFO is attracted towards an expected equilibrium, in which the different sources of unpifoness are balanced. However, there are also some limitations.

On the one hand, this equilibrium exists only in expectation, and the queue bounds are also only attracted to it in expectation. The actual queue bounds depend on the order in which packets arrive, as do inversions. So, even though on average, assuming a random rank ordering, the system might be balanced, there exist particular *adversarial* rank orderings, which "outplay" the two stages to create events of large unpifoness. An adversary might attempt to abuse this by coordinating a large number of flows to force an adversarial ordering of packet ranks. As an example, she might try to increase all queue bounds as much as possible before triggering a "push-down" reaction (e.g., by generating sequences of monotonically-increasing packet ranks). With the sudden decrease in queue-bound values, the high-rank packets mapped in the queues would generate inversions to the new packets.

Nevertheless, any non-malicious coexisting flow can easily thwart such strategies, by just randomly breaking the adversarial order. Still, it might be interesting to classify all adversarial orderings, and subsequently monitor the network to actively detect such type of attacks.

AccelTCP: Accelerating Network Applications with Stateful TCP Offloading

YoungGyoun Moon
KAIST

SeungEon Lee
KAIST

Muhammad Asim Jamshed
Intel Labs

KyoungSoo Park
KAIST

Abstract

The performance of modern key-value servers or layer-7 load balancers often heavily depends on the efficiency of the underlying TCP stack. Despite numerous optimizations such as kernel-bypassing and zero-copying, performance improvement with a TCP stack is fundamentally limited due to the *protocol conformance overhead* for compatible TCP operations. Unfortunately, the protocol conformance overhead amounts to as large as 60% of the entire CPU cycles for short-lived connections or degrades the performance of L7 proxying by 3.2x to 6.3x.

This work presents AccelTCP, a hardware-assisted TCP stack architecture that harnesses programmable network interface cards (NICs) as a TCP protocol accelerator. AccelTCP can offload complex TCP operations such as connection setup and teardown completely to NIC, which simplifies the host stack operations and frees a significant amount of CPU cycles for application processing. In addition, it supports running connection splicing on NIC so that the NIC relays all packets of the spliced connections with zero DMA overhead. Our evaluation shows that AccelTCP enables short-lived connections to perform comparably to persistent connections. It also improves the performance of Redis, a popular in-memory key-value store, and HAProxy, a widely-used layer-7 load balancer, by 2.3x and 11.9x, respectively.

1 Introduction

Transmission Control Protocol (TCP) [24] is undeniably the most popular protocol in modern data networking. It guarantees reliable data transfer between two end-points without overwhelming either end-point nor the network itself. It has become ubiquitous as it simply requires running on the Internet Protocol (IP) [23] that operates on almost every physical network.

Ensuring the desirable properties of TCP, however, often entails a severe performance penalty. This is especially pronounced with the recent trend that the gap between CPU capacity and network bandwidth widens. Two notable scenarios where modern TCP servers suffer from poor performance are handling short-lived connections and layer-7 (L7) proxying. Short-lived connections incur a serious overhead in processing small control packets while an L7 proxy requires large compute cycles and memory bandwidth for relaying packets between two connections. While recent kernel-bypass TCP stacks [5, 30, 41, 55, 61] have substantially improved the performance of short RPC transactions, they still need to track flow states whose computation cost is as large as 60% of the entire CPU cycles (Section §2). An alternative might be to adopt RDMA [37, 43] or a custom RPC protocol [44], but the former requires an extra in-network support [7, 8, 70] while the latter is limited to closed environments. On the other hand, an application-level proxy like L7 load balancer (LB) may benefit from zero copying (e.g., via the `splice()` system call), but it must perform expensive DMA operations that would waste memory bandwidth.

The root cause of the problem is actually clear – the TCP stack must maintain mechanical protocol conformance regardless of what the application does. For instance, a key-value server has to synchronize the state at connection setup and closure even when it handles only two data packets for a query. An L7 LB must relay the content between two separate connections even if its core functionality is determining the back-end server.

AccelTCP addresses this problem by exploiting modern network interface cards (NICs) as a TCP protocol accelerator. It presents a dual-stack TCP design that splits the functionality between a host and a NIC stack. The host stack holds the main control of all TCP operations;

it sends and receives data reliably from/to applications and performs control-plane operations such as congestion and flow control. In contrast to existing TCP stacks, however, it accelerates TCP processing by selectively offloading *stateful* operations to the NIC stack. Once offloaded, the NIC stack processes connection setup and teardown as well as connection splicing that relays packets of two connections entirely on NIC. The goal of AccelTCP is to extend the performance benefit of traditional NIC offload to short-lived connections and application-level proxying while being complementary to existing offloading schemes.

Our design brings two practical benefits. First, it significantly saves the compute cycles and memory bandwidth of the host stack as it simplifies the code path. Connection management on NIC simplifies the host stack as the host needs to keep only the established connections as well as it avoids frequent DMA operations for small control packets. Also, forwarding packets of spliced connections directly on NIC eliminates DMA operations and application-level processing. This allows the application to spend precious CPU cycles on its main functionality. Second, the host stack makes an offloading decision flexibly on a per-flow basis. When an L7 LB needs to check the content of a response of select flows, it opts them out of offloading while other flows still benefit from connection splicing on NIC. When the host stack detects overload of the NIC, it can opportunistically reduce the offloading rate and use the CPU instead.

However, performing stateful TCP operations on NIC is non-trivial due to following challenges. First, maintaining consistency of transmission control blocks (TCBs) across host and NIC stacks is challenging as any operation on one stack inherently deviates from the state of the other. To address the problem, AccelTCP always transfers the ownership of a TCB along with an offloaded task. This ensures that a single entity solely holds the ownership and updates its state at any given time. Second, stateful TCP operations increase the implementation complexity on NIC. AccelTCP manages the complexity in two respects. First, it exploits modern smart NICs equipped with tens of processing cores and a large memory, which allows flexible packet processing with C and/or P4 [33]. Second, it limits the complexity by resorting to a stateless protocol or by cooperating with the host stack. As a result, the entire code for the NIC stack is only 1,501 lines of C code and 195 lines of P4 code, which is small enough to manage on NIC.

Our evaluation shows that AccelTCP brings an enormous performance gain. It outperforms mTCP [41] by 2.2x to 3.8x while it enables non-persistent connections to perform comparably to persistent connections on IX [30] or mTCP. AccelTCP's connection splicing offload achieves a full line rate of 80 Gbps for L7 proxying of 512-byte messages with only a single CPU core. In terms of real-world applications, AccelTCP improves the performance of Redis [17] and HAProxy [6] by a factor of 2.3x and 11.9x, respectively.

The contribution of our work is summarized as follows. (1) We quantify and present the overhead of TCP protocol conformance in short-lived connections and L7 proxying. (2) We present the design of AccelTCP, a dual-stack TCP processing system that offloads select features of stateful TCP operations to NIC. We explain the rationale for our target tasks of NIC offload, and present a number of techniques that reduce the implementation complexity on smart NIC. (3) We demonstrate a significant performance benefit of AccelTCP over existing kernel-bypass TCP stacks like mTCP and IX as well as the benefit to real-world applications like a key-value server and an L7 LB.

2 Background and Motivation

In this section, we briefly explain the need for an NIC-accelerated TCP stack, and discuss our approach.

2.1 TCP Overhead in Short Connections & L7 Proxying

Short-lived TCP connections are prevalent in data centers [31, 65] as well as in wide-area networks [54, 64, 66]. L7 proxying is also widely used in middlebox applications such as L7 LBs [6, 36] and application-level gateways [2, 19]. Unfortunately, application-level performance of these workloads is often suboptimal as the majority of CPU cycles are spent on TCP stack operations. To better understand the cost, we analyze the overhead of the TCP stack operations in these workloads. To avoid the inefficiency of the kernel stack [38, 39, 60], we use mTCP [41], a scalable user-level TCP stack on DPDK [10], as our baseline stack for evaluation. We use one machine for a server (or proxy) and four clients and four back-end servers, all equipped with a 40GbE NIC. The detailed experimental setup is in Section §6.

Small message transactions: To measure the overhead of a short-lived TCP connection, we compare the performance of non-persistent vs. persistent connections with a large number of concurrent RPC transactions. We spawn 16k connections where each transaction exchanges one small request and one small response (64B) between a client and a server. A non-persistent connection performs only a single transaction while a persistent connection repeats the transactions without

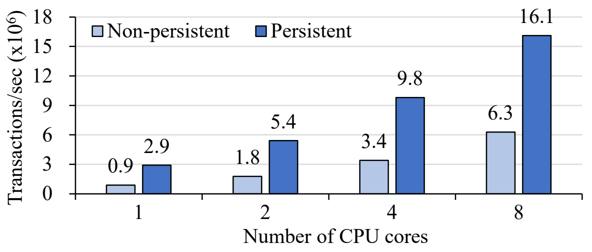


Figure 1: Small packet (64B) performance with non-persistent and persistent connections

a closure. To minimize the number of small packets, we patch mTCP to piggyback every ACK on the data packet.

Figure 1 shows that persistent connections outperform non-persistent connections by 2.6x to 3.2x. The connection management overhead is roughly proportional to the number of extra packets that it handles; two packets per transaction with a persistent connection vs. six [1] packets for the same task with a non-persistent connection. Table 1 shows the breakdown of the CPU cycles where almost 60% of them are attributed to connection setup and teardown. The overhead mainly comes from TCP protocol handling with connection table management, TCB construction and destruction, packet I/O, and L2/L3-level processing of control packets.

Our experiments may explain the strong preference to persistent connections in data centers. However, not all applications benefit from the persistency. When application data is inherently small or transferred sporadically [32, 69], it would result in a period of inactivity that taxes on server resources. Similarly, persistent connections are often deprecated in PHP applications to avoid the risk of resource misuse [28]. In general, supporting persistent connections is cumbersome and error-prone because the application not only needs to keep track of connection states, but it also has to periodically check connection timeout and terminate idle connections. By eliminating the connection management cost with NIC offload, our work intends to free the developers from this burden to choose the best approach without performance concern.

Application-level proxying: An L7 proxy typically operates by (1) terminating a client connection (2) accepting a request from the client and determining the back-end server with it, and creating a server-side connection, and (3) relaying the content between the client and the back-end server. While the key functionality of an L7 proxy is to map a client-side connection to a back-end server, it consumes most of CPU cycles on relaying the packets between the two connections. Packet

[1]SYN, SYN-ACK, ACK-request, response-FIN, FIN-ACK, and ACK.

Connection setup/ teardown	TCP processing and state update	24.0%	60.5%
	TCP connection state init/destroy	17.2%	
	Packet I/O (control packet)	10.2%	
	L2-L3 processing/forward	9.1%	
Message delivery	TCP processing and state update	11.0%	29.0%
	Message copy via socket buffer	8.4%	
	Packet I/O (data packet)	5.1%	
	L2-L3 processing/forward	4.5%	
Socket/epoll API calls			5.6%
Timer handling and context switching			3.5%
Application logic			1.4%

Table 1: CPU usage breakdown of a user-level TCP echo server (a single 64B packet exchange per connection)

	64B	1500B
L7 proxy (mTCP)	2.1 Gbps	5.3 Gbps
L7 proxy with splice() (mTCP)	2.3 Gbps	6.3 Gbps
L3 forward at host (DPDK)	7.3 Gbps	39.8 Gbps
L3 forward at NIC [2]	28.8 Gbps	40.0 Gbps

Table 2: L7 proxying and L3 forwarding performance on a single CPU core

relaying incurs a severe memory copying overhead as well as frequent context switchings between the TCP stack and the application. While zero-copying APIs like splice() can mitigate the overhead, DMA operations between the host memory and the NIC are unavoidable even with a kernel-bypass TCP stack.

Table 2 shows the 1-core performance of a simple L7 proxy on mTCP with 16k persistent connections (8k connections for clients-to-proxy and proxy-to-backend servers, respectively). The proxy exchanges n-byte (n=64 or 1500) packets between two connections, and we measure the wire-level throughput at clients including control packets. We observe that TCP operations in the proxy significantly degrade the performance by 3.2x to 6.3x compared to simple packet forwarding with DPDK [10], despite using zero-copy splice(). Moreover, DMA operations further degrade the performance by 3.8x for small packets.

Summary: We confirm that connection management and packet relaying consume a large amount of CPU cycles, severely limiting the application-level performance. Offloading these operations to NIC promises a large potential for performance improvement.

2.2 NIC Offload of TCP Features

There have been a large number of works and debates on NIC offloading of TCP features [35, 47, 50, 57]. While AccelTCP pursues the same benefit of saving CPU cycles

[2]All 120 flow-processing cores in Agilio LX are enabled.

and memory bandwidth, it targets a different class of applications neglected by existing schemes.

Partial TCP offload: Modern NICs typically support partial, fixed TCP function offloads such as TCP/IP checksum calculation, TCP segmentation offload (TSO), and large receive offload (LRO). These significantly save CPU cycles for processing large messages as they avoid scanning packet payload and reduce the number of interrupts to handle. TSO and LRO also improve the DMA throughput as they cut down the DMA setup cost required to deliver many small packets. However, their performance benefit is mostly limited to large data transfer as short-lived transactions deal with only a few of small packets.

Full Stack offload: TCP Offload Engine (TOE) takes a more ambitious approach that offloads entire TCP processing to NIC [34, 67]. Similar to our work, TOE eliminates the CPU cycles and DMA overhead of connection management. It also avoids the DMA transfer of small ACK packets as it manages socket buffers on NIC. Unfortunately, full stack TOE is unpopular in practice as it requires invasive modification of the kernel stack and the compute resource on NIC is limited [12]. Also, operational flexibility is constrained as it requires firmware update to fix bugs or to replace algorithms like congestion control or to add new TCP options. Microsoft's TCP Chimney [15] deviates from the full stack TOE as the kernel stack controls all connections while it offloads only data transfer to the NIC. However, it suffers from similar limitations that arise as the NIC implements TCP data transfer (e.g., flow reassembly, congestion and flow control, buffer management). As a result, it is rarely enabled these days [27].

In comparison, existing schemes mainly focus on efficient large data transfer, but AccelTCP targets performance improvement with short-lived connections and L7 proxying. AccelTCP is complementary to existing partial TCP offloads as it still exploits them for large data transfer. Similar to TCP Chimney, AccelTCP's host stack assumes full control of the connections. However, the main offloading task is completely the opposite: AccelTCP offloads connection management while the host stack implements entire TCP data transfer. This design substantially reduces the complexity on NIC while it extends the benefit to an important class of modern applications.

2.3 Smart NIC for Stateful Offload

Smart NICs [1, 3, 14, 25] are gaining popularity as they support flexible packet processing at high speed with programming languages like C or P4 [33]. Recent smart NICs are flexible enough to run Open vSwitch [62], Berkeley packet filter [49], or even key-value lookup [53], often achieving 2x to 3x performance improvement over CPU-based solutions [16]. In this work, we use Netronome Agilio LX as a smart NIC platform to offload stateful TCP operations.

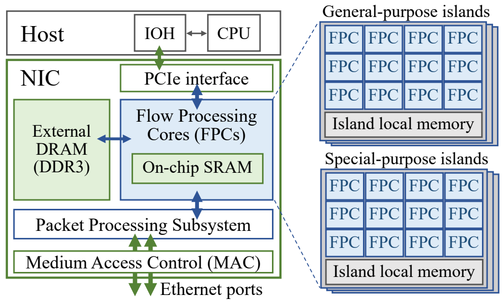


Figure 2: Architecture of SoC-based NIC (Agilio LX)

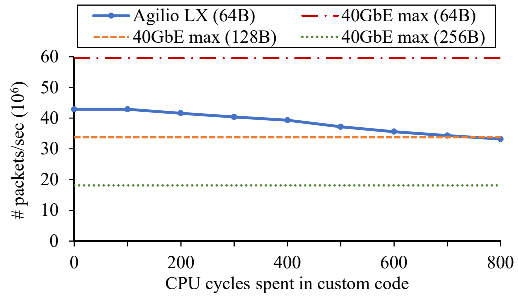


Figure 3: Packet forwarding performance on Agilio LX

As shown in Figure 2, Agilio LX employs 120 flow processing cores (FPCs) running at 1.2GHz. 36 FPCs are dedicated to special operations (e.g., PCI or Interlaken) while remaining 84 FPCs can be used for arbitrary packet processing programmed in C and P4. One can implement the basic forwarding path with a match-action table in P4 and add custom actions that require a fine-grained logic written in C. The platform also provides fast hashing, checksum calculation, and cryptographic operations implemented in hardware.

One drastic difference from general-purpose CPU is that FPCs have multiple layers of non-uniform memory access subsystem – registers and memory local to each FPC, shared memory for a cluster of FPCs called "island", or globally-accessible memory by all FPCs. Memory access latency ranges from 1 to 500 cycles depending on the location, where access to smaller memory tends to be faster than larger ones. We mainly use internal memory (IMEM, 8MB of SRAM) for flow metadata and external memory (EMEM, 8GB of DRAM) for packet contents. Depending on the flow metadata size, IMEM can support up to 128K to 256K concurrent flows. While EMEM would support more flows, it is 2.5x slower. Each FPC

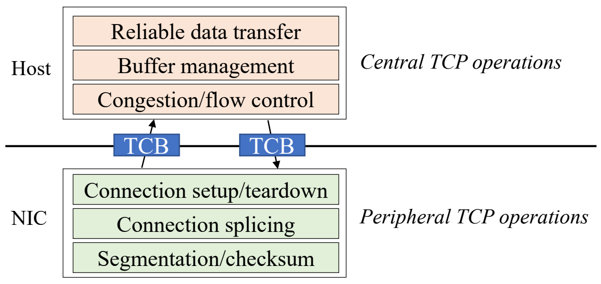


Figure 4: Split of TCP functionality in AccelTCP

can run up to 8 cooperative threads – access to slow memory by one thread would trigger a hardware-based context switch to another, which takes only 2 cycles. This hides memory access latency similarly to GPU.

Figure 3 shows the packet forwarding performance of Agilio LX as a function of cycles spent by custom C code, where L3 forwarding is implemented in P4. We see that it achieves the line rate (40 Gbps) for any packets larger than 128B. However, 64B packet forwarding throughput is only 42.9 Mpps (or 28.8 Gbps) even without any custom code. We suspect the bottleneck lies in scattering and gathering of packets across the FPCs. The performance starts to drop as the custom code spends more than 200 cycles, so minimizing cycle consumption on NIC is critical for high performance.

3 AccelTCP Design Rationale

AccelTCP is a dual-stack TCP architecture that harnesses NIC hardware as a TCP protocol accelerator. So, the primary task in AccelTCP's design is to determine the target for offloading. In this regard, AccelTCP divides the TCP stack operations into two categories: *central* TCP operations that involve application data transfer and *peripheral* TCP operations required for protocol conformance or mechanical operations that can bypass the application logic. Central TCP operations refer to all aspects of application data transfer – reliable data transfer with handling ACKs, inferring loss and packet retransmission, tracking received data and performing flow reassembly, enforcing congestion/flow control, and detecting errors (e.g., abrupt connection closure by a peer). These are typically complex and subject to flexible policies, which demands variable amount of compute cycles. One can optimize them by exploiting flow-level parallelism [5, 30, 41, 59] or by steering the tasks into fast and slow paths [48] on kernel-bypass stacks. However, the inherent complexity makes it a poor fit for NIC offloading as evidenced by the full stack TOE approach.

Peripheral operations refer to the remaining tasks whose operation is logically independent from the application. These include traditional partial NIC offload tasks [3], connection setup and teardown, and blind relaying of packets between two connections that requires no application-level intervention. Peripheral tasks are either stateless operations with a fixed processing cost or lightly stateful operations that synchronize the states for reliable data transfer. We mainly target these operations for offloading as they can be easily separated from the host side that runs applications.

Connection management offload: State synchronization at the boundary of a connection is a key requirement for TCP, but it is a pure overhead from the application's perspective. While NIC offload is logically desirable, conventional wisdom suggests otherwise due to complexity [15, 48]. Our position is that one can tame the complexity on recent smart NICs. First, connection setup operations can be made stateless with SYN-cookies [20]. Second, the common case of connection teardown is simple state transition, and modern smart NICs have enough resources to handle a few exceptions.

Connection splicing offload: Offloading connection splicing to NIC is conceptually complex as it requires state management of two separate connections on NIC. However, if the application does not modify the relayed content, as is often the case with L7 LBs, we can simulate a single logical connection with two physical connections. This allows the NIC to operate as a fast packet forwarder that simply translates the packet header. The compute cycles for this are fixed with a small per-splicing state.

To support the new offload tasks, we structure the dual-stack design with the following guidelines.

1. Full control by the host side: The host side should enforce full control of offloading, and it should be able to operate standalone. This is because the host stack must handle corner cases that cannot benefit from offload. For example, a SYN packet without the timestamp option should be handled by the host stack as SYN-cookie-based connection setup would lose negotiated TCP options (Section §4). Also, the host stack could decide to temporarily disable connection offload when it detects the overload of the NIC.

2. Single ownership of a TCB: AccelTCP offloads stateful operations that require updating the TCB. However, maintaining shared TCBs consistently across two stacks is very challenging. For example, a send buffer may have unacknowledged data along with the last FIN packet. The host stack may decide to deliver all data packets for itself while it offloads the connection teardown to NIC simultaneously. Unfortunately, handling

[3]Such as checksum calculation, TSO, and LRO.

ACKs and retransmission across two stacks require careful synchronization of the TCB. To avoid such a case, AccelTCP enforces an exclusive ownership of the TCB at any given time – either host or NIC stack holds the ownership but not both. In the above case, the host stack offloads the entire data to the NIC stack and forgets about the connection. The NIC stack handles remaining data transfer as well as connection teardown.

3. Minimal complexity on NIC: Smart NICs have limited compute resources, so it is important to minimize complex operations on NIC. A tricky case arises at connection teardown as the host stack can offload data transfer as well. In that case, the host stack limits the amount of data so that the NIC stack avoids congestion control and minimizes state tracking of data packets.

4 AccelTCP NIC Dataplane

In this section, we present the design of AccelTCP NIC stack in detail. Its primary role is to execute three offload tasks requested by the host stack. Each offload task can be enabled independently and the host side can decide which flows to benefit from it. The overall operation of NIC offload is shown in Figure 5.

4.1 Connection Setup Offload

An AccelTCP server can offload the connection setup process completely to the NIC stack. For connection setup offload, the server installs the metadata such as local IP addresses and ports for listening on NIC, and the NIC stack handles all control packets in a three-way handshake. Then, only the established connections are delivered to the host stack.

AccelTCP leverages SYN cookies [20] for stateless handshake on NIC. Stateless handshake enables a more efficient implementation as most smart NICs support fast one-way hashing functions in hardware [1, 3, 14]. When a SYN packet arrives, the NIC stack responds with an SYN-ACK packet whose initial sequence number (ISN) is chosen carefully. The ISN consists of 24 bits of a hash value produced with the input of the 4-tuple of a connection and a nonce, 3 bits of encoded maximum segment size (MSS), and time-dependent 5 bits to prevent replay attacks. When an ACK for the SYN-ACK packet arrives, the NIC stack verifies if the ACK number matches (ISN + 1). If it matches, the NIC stack passes the ACK packet up to the host stack with a special marking that indicates a new connection and the information on negotiated TCP options. To properly handle TCP options carried in the initial SYN, the NIC stack encodes all negotiated options in the TCP Timestamps option [22] of the SYN-ACK packet [9]. Then, the NIC stack can retrieve the information from the TSecr value echoed back with the ACK packet. In addition, we use extra one bit in the timestamp field to differentiate a SYN-ACK packet from other packets. This would allow the NIC stack to bypass ACK number verification for normal packets. The TCP Timestamps option is popular (e.g., enabled on 84% of hosts in a national-scale network [51]), and enabled by default on most OSes, but in case a client does not support it, the NIC stack hands the setup process over to the host stack.

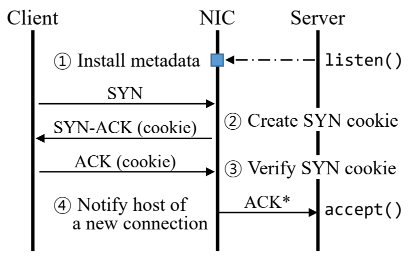
One case where SYN cookies are deprecated is when the server must send the data first after connection setup (e.g., SMTP server). In this case, the client could wait indefinitely if the client-sent ACK packet is lost as the SYN-ACK packet is never retransmitted. Such applications should disable connection setup offload and have the host stack handle connection setup instead.

4.2 Connection Teardown Offload

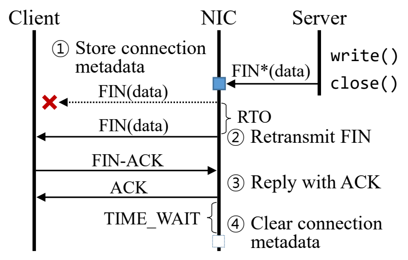
The application can ask for offloading connection teardown on a per-flow basis. If the host stack decides to offload connection teardown, it hands over the ownership of the TCB and remaining data in the send buffer to the NIC stack. Then, the host stack removes the flow entry from its connection table, and the NIC stack continues to handle the teardown.

Connection teardown offload is tricky as it must maintain per-flow states while it should ensure reliable delivery of the FIN packet with the offloaded data. To minimize the complexity, the host stack offloads connection teardown only when the following conditions are met. First, the amount of remaining data should be smaller than the send window size. This would avoid complex congestion control on NIC while it still benefits most short-lived connections. [4] Second, if the application wants to confirm data delivery at close(), the host stack should handle the connection teardown by itself. For example, an application may make close() to block until all data is delivered to the other side (e.g., *SO_LINGER* option). In that case, processing the teardown at the host stack is much simpler as it needs to report the result to the application. Fortunately, blocking close() is rare in busy TCP servers as it not only kills the performance, but a well-designed application-level protocol may avoid it. Third, the number of offloaded flows should not exceed a threshold, determined by available memory size on NIC. For each connection teardown, the host stack first checks the number of connection closures being handled by the NIC, and the host stack carries out the connection teardown if the number exceeds the threshold.

[4] RFC 6928 [21] suggests 10 MSS as the initial window size.



(a) Connection setup offload



(b) Connection teardown offload

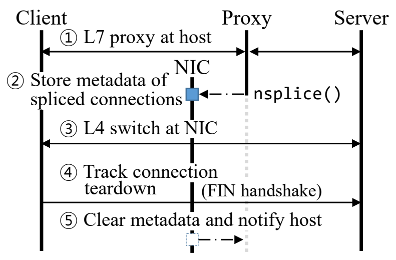
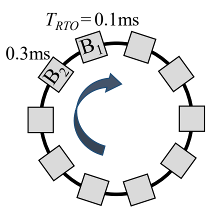
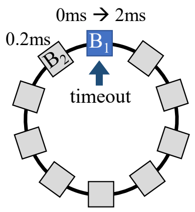

Client
Proxy
Server
① L7 proxy at host
NIC
② Store metadata of spliced connections
nsplice()
③ L4 switch at NIC
④ Track connection teardown
(FIN handshake)
⑤ Clear metadata and notify host
(c) Connection splicing offload

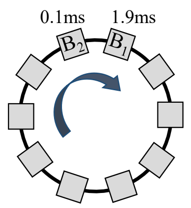
Figure 5: AccelTCP NIC offload (We show only active close() by server for (b), but it also supports passive close().)



(a) At T



(b) At $T + 0.1$ ms



(c) At T + 0.2 ms

Figure 6: Timer bitmap wheel for RTO management on NIC. T_{RTO} represents the remaining time until retransmission.

The NIC stack implements the teardown offload by extending the TSO mechanism. On receiving the offload request, it stores a 26-byte flow state [5] at the on-chip SRAM (e.g., 8MB of IMEM), segments the data into TCP packets, and send them out. Then, it stores the entire packets at the off-chip DRAM (e.g., 8GB of EMEM) for potential retransmission. This would allow tracking over 256k concurrent flows being closed on NIC.

Timeout management: The teardown process requires timeout management for packet retransmission and for observing a timeout in the TIME_WAIT state. AccelTCP uses three duplicate ACKs and expiration of retransmission timeout (RTO) as a trigger for packet retransmission. For teardown offload, however, RTO is the main mechanism as the number of data packets is often too small for three duplicate ACKs. Also, any side that sends the FIN first would end up in the TIME_WAIT state for a timeout. A high-performance server typically avoids this state by having the clients initiate the connection closure, but sometimes it is inevitable. AccelTCP supports the TIME_WAIT state, but it shares the same mechanism as RTO management for the timer.

Unfortunately, an efficient RTO implementation on NIC is challenging. For multicore CPU systems, a list or a hash table implementation would work well as each CPU core handles only its own flows affinitized to it without a lock. However, smart NICs often do not guarantee flow-core affinity, so a list-based implementation would incur huge lock contention with many processor cores.

We observe that RTO management is write-heavy as each offloaded flow (and each packet transmission) would register for a new RTO. Thus, we come up with a data structure called *timer bitmap wheel*, which allows concurrent updates with minimal lock contention. It consists of N timer bitmaps where each bitmap is associated with a distinct timeout value. The time interval between two neighboring timer bitmaps is fixed (e.g., 100 us for Figure 6). When one time interval elapses, all bitmaps rotate in the clockwise direction by one interval, like Figure 6-(b). Bitmap rotation is efficiently implemented by updating a pointer to the RTO-expired bitmap every time interval. Each timer bitmap records all flows with the same RTO value, where the location of a bit represents a flow id (e.g., n-th bit in a bitmap refers to a flow id, n). When the RTO of a timer bitmap expires, all flows in the bitmap retransmit their unacknowledged packets. From the location of each bit that is set, one can derive the corresponding flow id and find the pointer to its flow state that holds all the metadata required for retransmission. Then, all bits in the bitmap are reset to zero and its RTO is reset to (N x (time interval)). RTO-expired flows register for a new RTO. When an ACK for the FIN of a flow arrives, the flow is removed from its RTO bitmap. One can implement an RTO larger than the maximum by keeping a counter in the flow state that decrements every expiration of the maximum RTO.

The timer bitmap wheel allows concurrent updates by multiple flows as long as their flow ids belong to different 32-bit words in the bitmap. Only the flows whose ids share the same 32-bit word contend for a lock for access. On the down side, it exhibits two overheads: memory space for bitmaps and bitmap scanning at RTO expiration. The memory consumption is not a big concern as it requires only 8KB for each bitmap for 64k concurrent flows being closed. We reduce the scanning overhead by having multiple cores scan a different bitmap region in parallel. Keeping a per-region counter might further

[5] a 4-tuple of the connection, TCP state, expected sequence and ACK numbers, and current RTO.

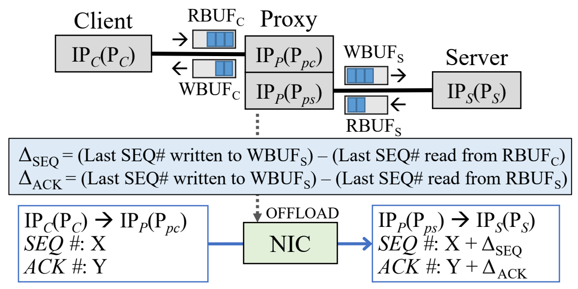


Figure 7: Connection splicing on NIC dataplane. IP_C, IP_P, IP_S: IP addresses of client, proxy, and server, P_C, P_S: port numbers of client and server, P_{pc}, P_{ps}: port numbers of proxy for the client side and the server side, $RBUF_C$, $RBUF_S$: read buffers on each side, $WBUF_C$, $WBUF_S$: write buffers on each side.

reduce the scanning overhead, but we find that the cost for counter update is too expensive even with atomic increment/decrement.

4.3 Connection Splicing Offload

Connection splicing offload on NIC allows zero-DMA data transfer. The key idea is to simulate a single connection by exploiting the NIC as a simple L4 switch that translates the packet header. An L7 proxy can ask for connection splicing on NIC if it no longer wants to relay packets of the two connections in the application layer. On a splicing offload request, the host stack hands over the states of two connections to NIC, and removes their TCBs from its connection table. The NIC stack takes over the ownership, and installs two L4 forwarding rules for relaying packets. The host stack keeps track of the number of spliced connections offloaded to NIC, and decides whether to offload more connections by considering the available memory on NIC.

Figure 7 shows the packet translation process. It simply swaps the 4-tuples of two connections and translates the sequence/ACK numbers and TCP/IP checksums of a packet with pre-calculated offsets. While the Figure assumes that the proxy does not modify any content, but one can easily support such a case. For example, if a proxy modifies request or response headers before splicing, the host stack only needs to reflect the extra delta in sequence and ACK numbers into the pre-calculated offsets. One limitation in our current scheme is that the proxy may not read or modify the packets any more after splicing offload.

Efficient TCP/IP checksum update: Translating a packet header requires TCP/IP checksum update. However, recalculating the TCP checksum is expensive as it scans the entire packet payload. To avoid the overhead, AccelTCP adopts *differential checksum update*, which exploits the fact that the one's complement addition is

```
1  On splicing offload for a flow from $IP_C(P_C)$ to $IP_S(P_S)$:
2      $CSO_{IP} \leftarrow IP_S + IP_C$
3      $CSO_{TCP} \leftarrow CSO_{IP} + P_S + P_{ps} - P_C - P_{pc} + \Delta_{SEQ} + \Delta_{ACK}$
4      **Store** $CSO_{IP}$ and $CSO_{TCP}$
5  For any next incoming packets from $IP_C(P_C)$ to $IP_S(P_S)$:
6      **Load** $CSO_{IP}$ and $CSO_{TCP}$
7      $CS_{IP} \leftarrow CS_{IP} + CSO_{IP}$
8      $CS_{TCP} \leftarrow CS_{TCP} + CSO_{TCP}$
9      If ($SEQ$ #) > $(-\Delta_{SEQ})$, then $CS_{TCP} \leftarrow CS_{TCP} - 1$
10     If ($ACK$ #) > $(-\Delta_{ACK})$, then $CS_{TCP} \leftarrow CS_{TCP} - 1$
```

Figure 8: Differential checksum update. CSO: checksum offset, CS: checksum. Other notations are in Figure 7. Note that + and − indicate 1's complement addition and subtraction.

both associative and distributive. Since only the 4-tuple of a connection and sequence and ACK numbers are updated, we only need to add the difference (or offset) of these values to the checksum. Figure 8 shows the algorithm. Upon splicing offload request, the NIC stack pre-calculates the offsets for IP and TCP checksums, respectively (Line 2-4). For each packet for translation, it adds the offsets to IP and TCP checksums, respectively (Line 7-8). One corner case arises if a sequence or an ACK number wraps around. In that case, we need to subtract 1 from the checksum to conform to 1's complement addition (Line 9-10).

Tracking teardown state: Since connection splicing operates by merging two connections into one, the NIC stack only needs to passively monitor connection teardown by the server and the client. When the spliced connection closes completely or if it is reset by any peer, the NIC stack removes the forwarding rule entries, and notifies the host stack of the closure. This allows reusing TCP ports or tracking connection statistics at the host.

5 AccelTCP Host Stack

The AccelTCP host stack is logically independent of the NIC stack. While our current implementation is based on mTCP [41], one can extend any TCP stack to harness our NIC offloading.

5.1 Socket API Extension

AccelTCP allows easy porting of existing applications by reusing the epoll()-based POSIX-like socket API of mTCP. In addition, it extends the API to support flexible NIC offloading as shown in Figure 9. First, AccelTCP adds extra socket options to `mtcp_setsockopt()` to enable connection setup and teardown offload to NIC. Note that the connection teardown offload request is advisory, so the host stack can decide not to offload the closure if the conditions are not met (Section §4.2). Second, AccelTCP adds `mtcp_nsplice()` to initiate splicing two connections on NIC. The host stack waits until all data

```
/* enable/disable setup and teardown offload
   - level  : IPPPROTO_TCP
   - optname: TCP_SETUP_OFFLOAD or TCP_TEARDOWN_OFFLOAD
   - optval : 1 (enable) or 0 (disable) */
int mtcp_setsockopt(mctx_t m, int sock, int level, int optname,
                    void *optval, socklen_t optlen);

/* offload connection splicing of two connections */
int mtcp_nsplice(mctx_t m, int sock_c, int sock_s, callback_t* cb);

/* notified upon a closure of spliced connections */
typedef void (*callback_t)(nsplice_meta_t * meta);
```

Figure 9: Socket API extension for AccelTCP

in the send buffer are acknowledged while buffering any incoming packets. Then, it installs forwarding rules onto NIC, sending the buffered packets after header translation. After calling this function, the socket descriptors should be treated as if they are closed in the application. Optionally, the application may specify a callback function to be notified when the spliced connections finish. Through the callback function, AccelTCP provides (i) remote addresses of the spliced connections, (ii) the number of bytes transferred after offloaded to NIC dataplane, and (iii) how the connections are terminated (e.g., normal teardown or reset by any peer).

5.2 Host Stack Optimizations

We optimize the host networking stack to accelerate small message processing. While these optimizations are orthogonal to NIC offload, they bring a significant performance benefit to short-lived connections.

Lazy TCB creation: A full TCB of a connection ranges from 400 to 700 bytes even on recent implementations [5, 41]. However, we find that many of the fields are unnecessary for short-lived connections whose message size is smaller than the initial window size. To avoid the overhead of a large TCB, AccelTCP creates the full TCB only when multiple transactions are observed. Instead, the host stack creates a small quasi-TCB (40 bytes) for a new connection. If the application closes the connection after a single write, the host stack offloads the teardown and destroys the quasi-TCB.

Opportunistic zero-copy: Recent high-performance TCP stacks [30, 61, 68] bypass the socket buffer to avoid extra memory copying. However, this often freezes the application-level buffer even after sending data, or overflows the host packet buffers if the application does not read the packets in a timely manner. AccelTCP addresses this problem by opportunistically performing a zero-copy I/O. When a stream of packets arrive in order, the application waiting for a read event will issue a read call. Then, the content of the packets is copied directly to the application buffer while any leftover is written to the receive socket buffer. When an application sends data on an empty socket buffer, the data is directly written to the host packet buffer for DMA'ing to NIC. Only when the host packet buffer is full, the data is written to the send socket buffer. Our scheme observes the semantics of standard socket operations, allowing easy porting of existing applications. Yet, this provides the benefit of zero-copying to most short-lived connections.

User-level threading: mTCP spawns two kernel-level threads: a TCP stack thread and an application thread on each CPU core. While this allows independent operations of the TCP thread (e.g., timer operations), it incurs a high context switching overhead. To address the problem, we modify mTCP to use cooperative user-level threading [13]. We find that this not only reduces the context switching overhead, but it also allows other optimizations like lazy TCB creation and opportunistic zero-copying.

6 Evaluation

We evaluate AccelTCP by answering following questions. First, does stateful TCP offloading and host stack optimizations demonstrate a high performance in a variety of workloads? (§6.1) Second, does it deliver the performance benefit to real-world applications? (§6.2) Finally, is the extra cost of a smart NIC justifiable? (§6.3)

Experiment setup: Our experimental setup consists of one server (or a proxy), four clients, and four back-end servers. The server machine has an Intel Xeon Gold 6142 @ 2.6GHz with 128 GB of DRAM and a dual-port Netronome Agilio LX 40GbE NIC (NFP-6480 chipset). Each client has an Intel Xeon E5-2640 v3 @ 2.6GHz, and back-end servers have a mix of Xeon E5-2699 v4 @ 2.2GHz and Xeon E5-2683 v4 @ 2.1GHz. The client and backend server machines are configured with Intel XL710-QDA2 40GbE NICs. All the machines are connected to a Dell Z9100-ON switch, configured to run at 40 GbE speed. For TCP stacks, we compare AccelTCP against mTCP [41] and IX [30]. All TCP stacks employ DPDK [10] for kernel-bypass packet I/O. Clients and back-end servers run mTCP patched to use cooperative user-level threading as AccelTCP. For IX experiments, we use two dual-port Intel X520-DA2 10GbE NICs, and enable all four ports bonded with a L3+L4 hash to balance the load as IX does not support 40GbE NICs. We verify that any single 10GbE port does not become the bottleneck based on port-level statistics at the switch. Hyperthreading is disabled for mTCP and AccelTCP, and enabled for IX when it improves the performance.

Our current prototype uses CRC32 to generate SYN cookies for connection setup. To prevent state explosion attacks, one needs to use a cryptographic hash function (such as MD5 or SHA2). Unfortunately, the API sup-

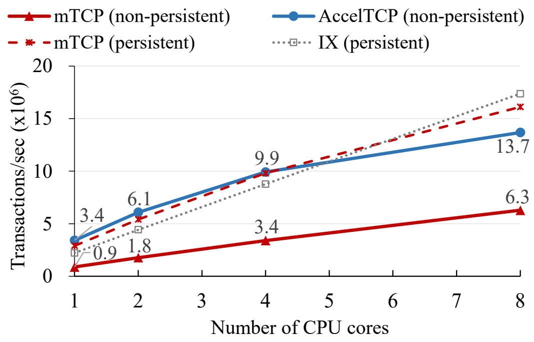


Figure 10: Throughputs of 64B packet transactions

Action	Mtps	Speedup
Baseline (w/o NIC offload)	0.89	1.0x
+ Enable setup offload (§4.1)	1.21	1.4x
+ Enable teardown offload (§4.2)	2.06	2.3x
+ Enable opportunistic TCB creation & opportunistic zero-copy (§5.2)	3.42	3.8x

Table 3: Breakdown of contribution by each optimization on a single CPU core (64B packet transactions)

port for hardware-assisted cryptographic operations in Agilio NICs is currently incomplete (for both C and P4 code), so we use CRC32 instead here.

6.1 Microbenchmark

We evaluate AccelTCP's performance for handling short-lived TCP connections and L7 proxying, and compare against the performance of the state-of-the-art TCP stacks: mTCP [41] and IX [30].

6.1.1 Short-lived Connection Performance

We evaluate the benefit of connection management offload by comparing the performance of TCP echo servers that perform 64B packet transactions with persistent vs. non-persistent connections. The TCP echo servers maintain 16k concurrent connections, and the performance results are averaged over one-minute period for five runs in each experiment. In the non-persistent case, a new connection is created immediately after every connection closure. AccelTCP offloads connection setup and offload to NIC while mTCP handles them using CPU. For IX, we evaluate only the persistent connection case as IX experiences a crash when handling thousands of concurrent connections with normal teardown.

Figure 10 compares the throughputs over varying numbers of CPU cores. AccelTCP achieves 2.2x to 3.8x better throughputs than non-persistent mTCP, comparable to those of persistent connections. Surprisingly, AccelTCP outperforms persistent connections by 13% to 54% for up to four CPU cores. This is because AccelTCP

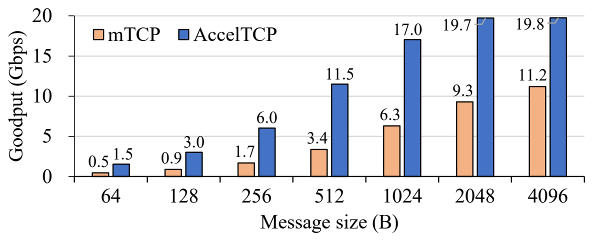


Figure 11: Performance of short-lived connections for varying message sizes on a single CPU core

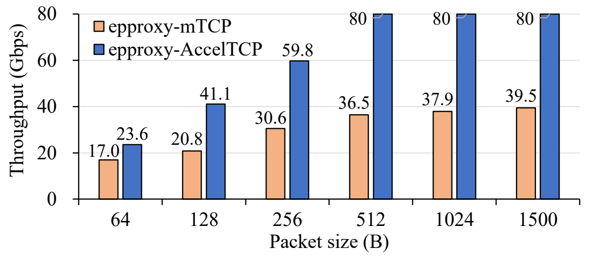


Figure 12: Comparison of L7 proxying throughputs

benefits from lazy TCB creation (§5.2) while persistent connections suffer from a CPU bottleneck. However, its eight-core performance is 22% lower than that of persistent IX, implying a bottleneck on NIC. Overall, connection management offload brings a significant performance benefit, which enables short-lived connections to perform comparably to persistent connections.

Table 3 shows the breakdown of performance in terms of the contribution by each optimization. We find that connection setup and teardown offload improve the baseline performance by 2.3x while other host stack optimizations contribute by extra 1.5x.

Figure 11 compares the goodputs over varying message sizes on a single CPU core. AccelTCP maintains the performance benefit over different message sizes with a speedup of 2.5x to 3.6x. The performance of messages larger than one MSS is limited at 20 Gbps, which seems impacted by our software TSO implementation on NIC. The current Agilio NIC SDK does not provide an API to exploit hardware TSO for programmable dataplane. We believe the single core performance would further improve with proper hardware support.

6.1.2 Layer-7 Proxing Performance

We now evaluate connection splicing offload with a simple L7 LB called epproxy that inspects the initial request, determines a back-end server, and relays the content between the client and the server. We measure the wire-level, receive-side throughput (including control packets) on the client side over different message sizes. Clients spawn 8k concurrent connections with epproxy, and the proxy creates 8k connections with back-

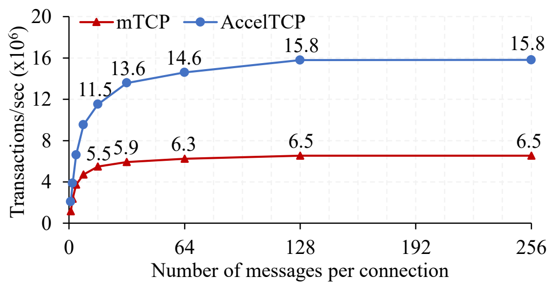


Figure 13: L7 proxying performance over varying numbers of message transactions per connection with 64B packets

end servers. We confirm that both clients and back-end servers are not the bottleneck. We configure epproxy-mTCP to use eight cores while epproxy-AccelTCP uses only a single core as CPU is not the bottleneck. All connections are persistent, and we employ both ports of the Agilio LX NIC here. The NIC is connected to the host via 8 lanes of PCIe-v3 [6].

Figure 12 shows that AccelTCP-proxy outperforms epproxy-mTCP by 1.4x to 2.2x even if the latter employs 8x more CPU cores. We make two observations here. First, the performance of epproxy-AccelTCP reaches full 80 Gbps from 512-byte messages, which exceeds the PCIe throughput of the NIC. This is because epproxy-AccelTCP bypasses host-side DMA and fully utilizes the forwarding capacity of the NIC. Second, epproxy-AccelTCP achieves up to twice as large goodput as the epproxy-mTCP. For example, epproxy-AccelTCP actually performs 2.8x more transactions per second than epproxy-mTCP for 64B messages. This is because AccelTCP splices two connections into a single one while mTCP relays two connections. For each request from a client, epproxy-mTCP must send an ACK as well as a response packet from the back-end server. In contrast, epproxy-AccelTCP replays only the response packet with a piggybacked ACK from the back-end server.

We move on to see if epproxy-AccelTCP fares well on non-persistent connections. Figure 13 shows the performance over varying numbers of message transactions per connection. AccelTCP performs 1.8x better at a single transaction, and the performance gap widens as large as 2.4x at 128 transactions per connection. This confirms that proxying non-persistent connections also benefit from splicing offload of AccelTCP.

6.2 Application Benchmark

We investigate if AccelTCP delivers the performance benefit to real-world applications.

Key-value store (Redis): We evaluate the effectiveness of AccelTCP with Redis (v.4.0.8) [17], a popular

[6]Theoretical maximum throughput is 63 Gbps according to [58].

	1-core	8-core
Redis-mTCP (kernel thread)	0.19 Mtps	1.38 Mtps
Redis-mTCP (user-level thread)	0.28 Mtps	1.94 Mtps
Redis-AccelTCP	0.44 Mtps	3.06 Mtps

Table 4: Redis performance for short-lived connections

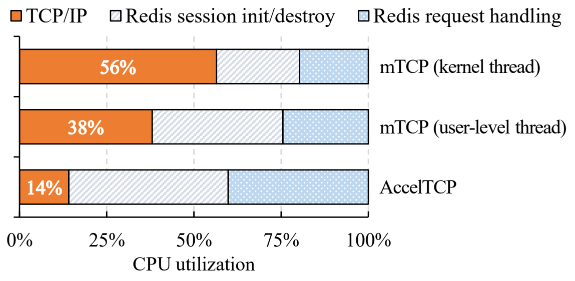


Figure 14: CPU breakdown of Redis on a single CPU core

in-memory key-value store. We use Redis on mTCP as a baseline server while we port it to use AccelTCP for comparison. We test with the USR workload from Facebook [29], which consists of 99.8% GET requests and 0.2% SET requests with short keys (< 20B) and 2B values. For load generation, we use a Redis client similar to memtier_benchmark [18] written in mTCP. We configure the Redis server and the clients to perform a single key-value transaction for each connection to show the behavior when short-lived connections are dominant.

Table 4 compares the throughputs. Redis-AccelTCP achieves 1.6x to 2.3x better performance than Redis-mTCP, and its performance scales well with the number of CPU cores. Figure 14 shows that mTCP consumes over a half of CPU cycles on TCP stack operations. In contrast, AccelTCP saves up to 75% of the CPU cycles for TCP processing. With AccelTCP, session initialization and destruction of Redis limits the performance. Our investigation reveals that the overhead mostly comes from dynamic memory (de)allocation (`zmalloc()` and `zfree()`) for per-connection metadata, which incurs a severe penalty for handling short-lived connections.

L7 LB (HAProxy): We see if AccelTCP improves the performance of HAProxy (v.1.8.0) [6], a widely used HTTP-based L7 LB. We first port HAProxy to use mTCP and AccelTCP, respectively, and evaluate the throughput with the SpecWeb2009[26]-like workload. The workload consists of static files whose size ranges from 30 to 5,670 bytes with an average file size of 728 bytes. For a fair comparison, we disable any header rewriting in the both version after delivering the first HTTP request. We spawn 8k persistent connections, using simple epoll-based clients and back-end servers running on mTCP. Table 5 compares the throughputs with with 1 core and

	1-core	8-core
HAProxy-mTCP	4.3 Gbps	6.2 Gbps
HAProxy-AccelTCP	73.1 Gbps	73.1 Gbps

Table 5: L7 LB performance for SpecWeb2009-like workload

	E5-2650v2	Gold 6142
mTCP (XL710-QDA2)	1.00	1.25
AccelTCP (Agilio LX)	1.93	1.96

Table 6: Comparison of normalized performance-per-dollar

8 cores. HAProxy-AccelTCP achieves 73.1 Gbps, a 11.9x better throughput than HAProxy-mTCP. The average response time of HAProxy-AccelTCP is 0.98 ms, 13.6x lower than that of HAProxy-mTCP. We observe that the performance benefit is much larger than in Section 6.1.2 because HAProxy has a higher overhead in application-level request processing and packet relaying.

6.3 Cost-effectiveness Analysis

AccelTCP requires a smart NIC, which is about 3-4x more expensive than a normal NIC at the moment. For fairness, we try comparing the cost effectiveness by the performance-per-dollar metric. We draw hardware prices from Intel [11] and Colfax [4] pages (as of August 2019), and use the performance of 64B packet transactions on short-lived connections. Specifically, we compare the performance-per-dollar with a system that runs mTCP with a commodity NIC (Intel XL710-QDA2, $440) vs. another system that runs AccelTCP with a smart NIC (Agilio LX, $1,750). For CPU, we consider Xeon E5-2650v2 ($1,170) and Xeon Gold 6142 ($2,950). For simplicity, we only consider CPU and NIC as hardware cost. Table 6 suggests that NIC offload with AccelTCP is 1.6x to 1.9x more cost-effective, and the gap would widen further if we add other fixed hardware costs.

7 Related Work

Kernel-bypass TCP stacks: Modern kernel-bypass TCP stacks such as mTCP [41], IX [30], SandStorm [55], F-Stack [5] deliver high-performance TCP processing of small message transactions. Most of them employ a fast user-level packet I/O [10], and exploit high parallelism on multicore systems by flow steering on NIC. More recently, systems like ZygOS [63], Shinjuku [42], and Shenango [59] further improve kernel-bypass stack by reducing the tail latency, employing techniques like task stealing, centralized packet distribution, and dynamic core reallocation. We believe that these works are largely orthogonal but complementary to our work as AccelTCP would enhance these stacks by offloading connection management tasks to NIC.

NIC offload: Existing TCP offloads mostly focus on improving large message transfer either by offloading the whole TCP stack [50] or by selectively offloading common send-receive operations [46]. In contrast, our work focuses on connection management and proxying whose performance is often critical to modern network workloads, while we intentionally avoid the complexity of application data transfer offloading. UNO [52] and Metron [45] strive to achieve optimal network function (NF) performance with NIC offload based on runtime traffic statistics. We plan to explore dynamic offloading of a subset of networking stack features (or connections) in response to varying load in the future. To offload TCP connection management, any L2-L4 NFs that should run prior to TCP stack (e.g., firewalling or host networking) must be offloaded to NIC accordingly. Such NFs can be written in P4 [40, 45, 56] and easily integrated with AccelTCP by properly placing them at ingress/egress pipelines of the NIC dataplane.

L7 proxing and short RPCs: Our connection splicing is inspired by the packet tunneling mechanism of Yoda L7 LB [36]. However, Yoda operates as a packet-level translator without a TCP stack, so it cannot modify any of relayed content. In contrast, an AccelTCP application can initiate the offload after any content modification. Also, AccelTCP packet translation runs on NIC hardware, promising better performance. Finally, we note that eRPC [44] achieves 5.0 Mtps RPC performance (vs. 3.4 Mtps of AccelTCP) on a single core. However, ePRC is limited to data center environments while AccelTCP is compatible to TCP and accommodates any TCP clients.

8 Conclusion

In this paper, we have presented AccelTCP that harnesses modern programmable NICs as a TCP protocol accelerator. Drawing the lessons from full stack TOE, AccelTCP's design focuses on minimizing the interaction with the host stack by offloading only select features of stateful TCP operations. AccelTCP manages the complexity on NIC by stateless handshake, single ownership of a TCB, and conditional teardown offload. In addition, it simplifies connection splicing by efficient packet header translation. We have also presented a number of optimizations that significantly improve the host stack.

We have demonstrated that AccelTCP brings a substantial performance boost to short-message transactions and L7 proxying. AccelTCP delivers a 2.3x speedup to Redis on a kernel-bypass stack while it improves the performance of HAProxy by a factor of 11.9. AccelTCP is available at `https://github.com/acceltcp`, and we hope our effort will renew the interest in selective NIC offload of stateful TCP operations.

Acknowledgments

We would like to thank our shepherd Andrew Moore and anonymous reviewers of NSDI 2020 for their insightful comments and feedback on the paper. We also thank Ilwoo Park for applying user-level threading to the mTCP stack. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (2018-0-00693, Development of an ultra low-latency user-level transfer protocol and 2016-0-00563, Research on Adaptive Machine Learning Technology Development for Intelligent Autonomous Digital Companion) as well as the SK-Hynix grant for Storage Media Solutions Center.

Appendix A. Host-NIC Communication Interface

The host and NIC stacks communicate with each other by piggybacking control information in the normal packets most time. It encodes the type of offload as unused EtherType values in the Ethernet header, and tags along other information in the special header between the Ethernet and IP headers.

Connection setup: When an application listens on a socket whose setup offload option is enabled, the host stack sends a special control packet to NIC, carrying the listening address/port and TCP options that must be delivered to the remote host during connection setup (e.g., MSS, Window scale factor, Selective ACK, etc.). To notify a new connection, the NIC stack sets the Ethertype of the ACK packet to 0x090A, and delivers the negotiated options in the TCP Timestamps option. The host stack extracts only the TCP options, and ignores the NIC-generated timestamp value.

Connection teardown: For teardown offload, the host stack creates a TSO packet that holds all remaining data in the send buffer, and sets the EtherType to 0x090B. It also encodes other information such as MSS (2 bytes), current RTO (4 bytes), and current TCP state (2 bytes) in the special header area. The NIC stack notifies the host stack of the number of connections being closed on NIC by either sending a control packet or tagging at any packet delivered to host.

Connection splicing: For splicing offload, the host stack uses 0x090C as EtherType, and writes the sequence and ACK number offsets (4 bytes each), and a 4-tuple of a connection in the special header. When the splicing offload packet is passed to the NIC stack, a race condition may arise if some packets in the flows are passed up to the host stack at the same time. To ensure correct forwarding, the host stack keeps the connection entries until it is notified that the splicing rules are installed at NIC. For reporting a closure of spliced connections, NIC creates a special control packet holding the connection information and traffic statistics with the EtherType, 0x090D, and sends it up to the host stack. By monitoring those control packets, the host stack can keep track of the number of active spliced connections on NIC.

References

[1] Agilio® LX 2x40GbE SmartNIC. https://www.netronome.com/m/documents/PB_Agilio_LX_2x40GbE.pdf. Accessed: 2019-08-27.

[2] Amazon API Gateway. https://aws.amazon.com/api-gateway/. Accessed: 2019-08-27.

[3] Cavium LiquidIO® II Network Appliance Smart NIC. https://www.marvell.com/documents/konmn48108xfxalr96jk/. Accessed: 2019-08-27.

[4] Colfax Direct. https://colfaxdirect.com. Accessed: 2019-08-27.

[5] F-Stack | High Performance Network Framework Based on DPDK. http://www.f-stack.org/. Accessed: 2019-08-27.

[6] HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/. Accessed: 2019-08-27.

[7] IEEE 802.1Qau – Congestion Notification. https://1.ieee802.org/dcb/802-1qau/. Accessed: 2019-08-27.

[8] IEEE 802.1Qbb - Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/. Accessed: 2019-08-27.

[9] Improving Syncookies. https://lwn.net/Articles/277146/. Accessed: 2019-08-27.

[10] Intel DPDK: Data Plane Development Kit. http://dpdk.org/. Accessed: 2019-08-27.

[11] Intel Product Specification. https://ark.intel.com. Accessed: 2019-08-27.

[12] Linux and TCP Offload Engines. https://lwn.net/Articles/148697/. Accessed: 2019-08-27.

[13] lthread: Multicore / Multithread Coroutine Library. https://lthread.readthedocs.io. Accessed: 2019-08-27.

[14] Mellanox BlueField™ SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf. Accessed: 2019-08-27.

[15] Microsoft Windows Scalable Networking Initiative. http://download.microsoft.com/download/5/b/5/5b5bec17-ea71-4653-9539-204a672f11cf/scale.doc. Accessed: 2019-08-27.

[16] Open vSwitch Offload and Acceleration with Agilio SmartNICs. https://www.netronome.com/m/redactor_files/WP_OVS_Benchmarking.pdf. Accessed: 2019-08-27.

[17] Redis. https://redis.io/. Accessed: 2019-08-27.

[18] RedisLabs/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark. Accessed: 2019-08-27.

[19] RFC 2663. https://tools.ietf.org/html/rfc2663. Accessed: 2019-08-27.

[20] RFC 4987. https://tools.ietf.org/html/rfc4987. Accessed: 2019-08-27.

[21] RFC 6928. https://tools.ietf.org/html/rfc6928. Accessed: 2019-08-27.

[22] RFC 7323. https://tools.ietf.org/html/rfc7323. Accessed: 2019-08-27.

[23] RFC 791. https://tools.ietf.org/html/rfc791. Accessed: 2019-08-27.

[24] RFC 793. https://tools.ietf.org/html/rfc793. Accessed: 2019-08-27.

[25] Solarflare SFA7942Q with Stratix V A7 FPGA. https://solarflare.com/wp-content/uploads/2018/11/SF-114649-CD-LATEST_Solarflare_AOE_SFA7942Q_Product_Brief.pdf. Accessed: 2019-08-27.

[26] SpecWeb2009 Benchmark. https://www.spec.org/web2009/. Accessed: 2019-08-27.

[27] Why Are We Deprecating Network Performance Features (KB4014193)? https://blogs.technet.microsoft.com/askpfeplat/2017/06/13/. Accessed: 2019-08-27.

[28] Why persistent connections are bad. https://meta.wikimedia.org/wiki/Why_persistent_connections_are_bad#Why_persistent_connections_are_bad. Accessed: 2019-08-27.

[29] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[30] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Transactions on Computer Systems (TOCS)*, 34(4):11, 2017.

[31] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 2010 ACM Internet Measurement Conference (IMC '10)*, 2010.

[32] Theophilus Benson, Aditya Akella, and David A Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC '10)*, 2010.

[33] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[34] Hsin-Chieh Chiang, Yuan-Pang Dai, and Chuei-Yu Wang. Full Hardware Based TCP/IP Traffic Offload Engine (TOE) Device and the Method Thereof, January 12 2010. US Patent 7,647,416.

[35] Douglas Freimuth, Elbert C Hu, Jason D LaVoie, Ronald Mraz, Erich M Nahum, Prashant Pradhan, and John M Tracey. Server Network Scalability and TCP Offload. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '05)*, 2005.

[36] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. Yoda: A Highly Available Layer-7 Load Balancer. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*, 2016.

[37] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*, 2016.

[38] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: A GPU-Accelerated Software Router. In *Proceedings of the 2010 ACM SIGCOMM Conference (SIGCOMM '10)*, 2010.

[39] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, 2012.

[40] David Hancock and Jacobus Van der Merwe. Hyper4: Using P4 to Virtualize the Programmable Data Plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies (CoNEXT '16)*, 2016.

[41] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[42] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[43] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[44] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[45] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI '18)*, 2018.

[46] Antoine Kaufmann. Efficient, Secure, and Flexible High Speed Packet Processing for Data Centers. In *PhD Thesis, University of Washington*, 2018.

[47] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas E. Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, 2016.

[48] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys '19)*, 2019.

[49] Jakub Kicinski and Nicolaas Viljoen. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. *Proceedings of Netdev 1.2, The Technical Conference on Linux Networking*, 1, 2016.

[50] Hyong-youb Kim and Scott Rixner. Connection Handoff Policies for TCP Offload Network Interfaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[51] Mirja Kühlewind, Sebastian Neuner, and Brian Trammell. On the state of ECN and TCP options on the Internet. In *Proceedings of the 14th Passive and Active Measurement Conference (PAM '13)*, 2013.

[52] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC '17)*, 2017.

[53] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[54] Gregor Maier, Anja Feldmann, Vern Paxson, and Mark Allman. On Dominant Characteristics of Residential Broadband Internet Traffic. In *Proceedings of the 2009 ACM Internet Measurement Conference (IMC '09)*, 2009.

[55] Ilias Marinos, Robert NM Watson, and Mark Handley. Network Stack Specialization for Performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.

[56] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM '17)*, 2017.

[57] Jeffrey C Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS '03)*, 2003.

[58] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM '18)*, 2018.

[59] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[60] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*, 2012.

[61] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):11, 2016.

[62] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, 2015.

[63] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[64] Lin Quan and John Heidemann. On the Characteristics and Reasons of Long-lived Internet Flows. In *Proceedings of the 2010 ACM Internet Measurement Conference (IMC '10)*, 2010.

[65] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM '15)*, 2015.

[66] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proceedings of the 11th International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*, 2013.

[67] Zhong-Zhen Wu and Han-Chiang Chen. Design and Implementation of TCP/IP Offload Engine System over Gigabit Ethernet. In *Proceedings of the 15th International Conference on Computer Communications and Networks*. IEEE, 2006.

[68] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC '16)*, 2016.

[69] Tao Zhang, Jianxin Wang, Jiawei Huang, Jianer Chen, Yi Pan, and Geyong Min. Tuning the Aggressive TCP Behavior for Highly Concurrent HTTP Connections in Intra-datacenter. *IEEE/ACM Transactions on Networking (TON)*, 25(6):3808–3822, 2017.

[70] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.

Enabling Programmable Transport Protocols in High-Speed NICs

Mina Tahmasbi Arashloo, *Princeton University*
Alexey Lavrov, *Princeton University*
Manya Ghobadi, *MIT*
Jennifer Rexford, *Princeton University*
David Walker, *Princeton University*
David Wentzlaff, *Princeton University*

Abstract

Data-center network stacks are moving into hardware to achieve 100 Gbps data rates and beyond at low latency and low CPU utilization. However, hardwiring the network stack in the NIC would stifle innovation in transport protocols. In this paper, we enable programmable transport protocols in high-speed NICs by designing Tonic, a flexible hardware architecture for transport logic. At 100 Gbps, transport protocols must generate a data segment every few nanoseconds using only a few kilobits of per-flow state on the NIC. By identifying common patterns across transport logic of different transport protocols, we design an efficient hardware "template" for transport logic that satisfies these constraints while being programmable with a simple API. Experiments with our FPGA-based prototype show that Tonic can support the transport logic of a wide range of protocols and meet timing for 100 Gbps of back-to-back 128-byte packets. That is, every 10 ns, our prototype generates the address of a data segment for one of more than a thousand active flows for a downstream DMA pipeline to fetch and transmit a packet.

1 Introduction

Transport protocols, along with the rest of the network stack, traditionally run in software. Despite efforts to improve their performance and efficiency [1,6,25,32], software network stacks tend to consume 30-40% of CPU cycles to keep up with applications in today's data centers [25,32,38].

As data centers move to 100 Gbps Ethernet, the CPU utilization of software network stacks becomes increasingly prohibitive. As a result, multiple vendors have developed hardware network stacks that run entirely on the network interface card (NIC) [8,10]. However, there are only two main transport protocols implemented on these NICs, both hardwired and modifiable only by the vendors:

RoCE. RoCE is used for Remote Direct Memory Access (RDMA) [8], using DCQCN [43] for congestion control and a simple go-back-N method for reliable data delivery.

TCP. A few vendors offload a TCP variant of their choice to the NIC to either be used directly through the socket API (TCP Offload Engine [10]) or to enable RDMA (iWARP [7]).

These protocols, however, only use a small fixed set out of the myriad of possible algorithms for reliable delivery [16,21,24,27,33,34] and congestion control [12,17,19,35,42,43] proposed over the past few decades. For instance, recent work suggests that low-latency data-center networks can significantly benefit from receiver-driven transport protocols [21,24,36], which is not an option in today's hardware stacks. In an attempt to deploy RoCE NICs in Microsoft data centers, operators needed to modify the data delivery algorithm to avoid livelocks in their network but had to rely on the NIC vendor to make that change [22]. Other algorithms have been proposed to improve RoCE's simple reliable delivery algorithm [31,34]. The long list of optimizations in TCP from years of deployment in various networks is a testament to the need for programmability in transport protocols.

In this paper, we investigate *how to make hardware transport protocols programmable*. Even if NIC vendors open up interfaces for programming their hardware, it takes a significant amount of expertise, time, and effort to implement transport protocols in high-speed hardware. To keep up with 100 Gbps, the transport protocol should generate and transmit a packet *every few nanoseconds*. It should handle *more than a thousand active flows*, typical in today's data-center servers [15, 37, 38]. To make matters worse, NICs are *extremely constrained* in terms of the amount of their on-chip memory and computing resources [30, 34].

We argue that transport protocols on high-speed NICs can be made programmable *without* exposing users to the full complexity of programming for high-speed hardware. Our argument is grounded in two main observations:

First, programmable *transport logic* is the key to enabling flexible hardware transport protocols. An implementation of a transport protocol performs several functionality such as connection management, data buffer management, and data transfer. However, its central responsibility, where most of the innovation happens, is to decide which data segments to transfer (data delivery) and when (congestion control), which we collectively call the *transport logic*. Thus, the key to programmable transport protocols on high-speed NICs is enabling users to modify the transport logic.

Second, we can exploit common patterns in transport logic to create reusable high-speed hardware modules. Despite their differences in application-level API (e.g., sockets and byte-stream abstractions for TCP vs. the message-based Verbs API for RDMA), and in connection and data buffer management, transport protocols share several common patterns. For instance, different transport protocols use different algorithms to detect lost packets. However, once a packet is declared lost, reliable transport protocols prioritize its retransmission over sending a new data segment. As another example, in congestion control, given the parameters determined by the control loop (e.g., congestion window and rate), there are only a few common ways to calculate how many bytes a flow can transmit at any time. This enables us to design an efficient "template" for transport logic in hardware that can be programmed with a simple API.

Using these insights, we design and develop Tonic, a programmable hardware architecture that can realize the *transport logic* of a broad range of transport protocols, using a simple API, while supporting 100 Gbps data-rates. Every clock cycle, Tonic generates the address of the next segment for transmission. The data segment is fetched from memory by a downstream DMA pipeline and turned into a full packet by the rest of the hardware network stack (Figure 1).

We envision that Tonic would reside on the NIC, replacing the hard-coded transport logic in hardware implementations of transport protocols (e.g., future RDMA NICs and TCP offload engines). Tonic provides a unified programmable architecture for transport logic, independent of how specific implementations of different transport protocols perform connection and data buffer management, and their application-level APIs. We will, however, describe how Tonic interfaces with the rest of the transport layer in general (§2) and how it can be integrated into Linux Kernel to interact with applications using socket API as an example (§5).

We implement a Tonic prototype in ∼8K lines of Verilog code and demonstrate Tonic's programmability by implementing the transport logic of a variety of transport protocols [13, 16, 23, 24, 34, 43] in less than 200 lines of code. We also show, using an FPGA, that Tonic meets timing for ∼100 Mpps, i.e., supporting 100Gbps of back-to-back 128B packets. That is, every 10ns, Tonic can generate the transport metadata required for a downstream DMA pipeline to fetch and send one packet. From generation to transmission, the latency of a single segment address through Tonic is ∼ 0.1µs, and Tonic can support up to 2048 concurrent flows.

2 Tonic as the Transport Logic

This section is an overview of how Tonic fits into the transport layer (§2.1), and how it overcomes the challenges of implementing transport logic on high-speed NICs (§2.2).

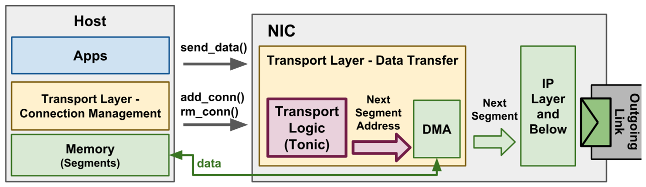


Figure 1: Tonic providing programmable transport logic in a hardware network stack on the NIC (sender-side).

2.1 How Tonic Fits in the Transport Layer

Sitting between applications and the rest of the stack, transport-layer protocols perform two main functions:
Connection Management includes creating and configuring endpoints (e.g., sockets for TCP and queue-pairs for RDMA) and establishing the connection in the beginning, and closing the connection and releasing its resources at the end.
Data Transfer involves delivering data from one endpoint to another, reliably and efficiently, in a stream of segments [1]. Different transport protocols provide different APIs for applications to request data transfer: TCP offers the abstraction of a byte-stream to which applications can continuously append data, while in RDMA, each "send" call to a queue-pair creates a separate work request and is treated as a separate message. Moreover, specifics of managing applications' data buffers differ across different implementations of transport protocols. Regardless, the transport protocol must deliver the outstanding data to its destination in multiple data segments that fit into individual packets. *Deciding which bytes comprise the next segment and when it is transmitted* is done by data delivery and congestion control algorithms, which we collectively call **transport logic** and implement in Tonic.

Figure 1 shows a high-level overview of how Tonic fits in a hardware network stack. To decouple Tonic from specifics of connection management and application-level APIs, connection setup and tear-down run outside of Tonic. Tonic relies on the rest of the transport layer to provide it with a unique identifier (*flow id*) for each established connection, and to explicitly add and remove connections using these IDs.

For data transfer on the sender side, Tonic keeps track of the number of outstanding bytes and transport-specific metadata to implement the transport logic, i.e., *generate the address of the next data segment* for each flow at the time designated by the congestion control algorithm. Thus, Tonic does not need to store and/or handle actual data bytes; it relies on the rest of the transport layer to manage data buffers on the host, DMA the segment whose address is generated in Tonic from memory, and notify it of new requests for data transmission on existing connections (see §5 for details).

The receiver-side of transport logic mainly involves generating control signals such as acknowledgments, per-packet

[1]We focus on reliable transport as it is more commonly used and more complicated to implement.

#	Observation	Examples
1	Only track a limited window of segments	TCP, NDP, IRN
2	Only keep a few bits of state per segment	TCP, NDP, IRN, RoCEv2
3	Lost segments first, new segments next	TCP, NDP, IRN, RoCEv2
4	Loss detection: Acks and timeouts	TCP, NDP, IRN
5	The three common credit calculation patterns: window, rate, and grant tokens	TCP, RoCEv2, NDP

Table 1: Common transport logic patterns.

grant tokens [21, 24, 36], or periodic congestion notification packets (CNPs) [43], while the rest of the transport layer manages receive data buffers and delivers the received data to applications. While handling received data can get quite complicated, generating control signals on the receiver is typically simpler than the sender. Thus, although we mainly focus on the sender, we reuse modules from the sender to implement a receiver solely for generating per-packet cumulative and selective acks and grant tokens at line rate.

2.2 Hardware Design Challenges

Implementing transport logic at line rate in the NIC is challenging due to two main constraints:

Timing constraints. The median packet size in data centers is less than 200 bytes [15, 37]. To achieve 100 Gbps for these small packets, the NIC has to send a packet every $\sim$10 ns. Thus, every $\sim$10 ns, the transport logic should determine which active flow should transmit which data segment next. To make this decision, it uses some state per flow (e.g., acknowledged data segments, duplicate acks, rate/window size, etc.) which is updated when various transport events happen (e.g., receiving an acknowledgment or a timeout). These updates could involve operations with non-negligible hardware overhead, such as searching bitmaps and arrays.

To allow for more time in processing each event while still determining the next data segment every $\sim$10 ns, we could conceivably pipeline the processing of transport events across multiple stages. However, pipelining is more tractable when incoming events are from *different* flows as they update different states. Processing back-to-back events for the *same* flow (e.g., generating data segments while receiving acknowledgments) requires updates to the same state, making it difficult to pipeline event processing while ensuring state consistency. Thus, we strive to process each transport event within 10 ns instead to quickly consolidate the state for the next event in case it affects the same flow.

Memory constraints. A typical data-center server has more than a thousand concurrent active flows with kilobytes of in-flight data [15, 37, 38]. Since NICs have just a few megabytes of high-speed memory [30, 34], the transport protocol can store only a few kilobits of state per flow on NIC.

Tonic's goal is to satisfy these tight timing and memory constraints while supporting programmability with a simple API. To do so, we identify common patterns across transport logic in various protocols that we implement as reusable fixed-function modules. These patterns allow us to optimize these modules for timing and memory, while simplifying the programming API by reducing the functionality users must specify. These patterns are summarized in Table 1, and are discussed in detail in next section, where we describe Tonic's components and how these patterns affect their design.

3 Tonic Architecture

Transport logic at the sender is what determines, for each flow, which data segments to transfer (data delivery) and when (congestion control). Conceptually, congestion control algorithms perform *credit management*, i.e., determine how many bytes a given flow can transmit at a time. Data delivery algorithms perform *segment selection*, i.e., decide which contiguous sequence of bytes a particular flow should transmit. Although the terms "data delivery" and "congestion control" are commonly associated with TCP-based transport protocols, Tonic provides a general programmable architecture for transport logic that can be used for other kinds of transport protocols as well, such as receiver-driven [21, 24, 36] and RDMA-based [8] transport protocols.

Tonic exploits the natural functional separation between data delivery and credit management to partition them into two components with separate state (Figure 2). The data delivery engine processes events related to generating, tracking, and delivery of segments, while the credit engine processes events related to adjusting each flow's credit and sending out segment addresses for those with sufficient credit.

At the cost of lightweight coordination between the two engines, this partitioning helps Tonic meet its timing constraints while concurrently processing multiple events (e.g., receipt of acknowledgments and segment transmission) every cycle. These events must read the current state of their corresponding flow, update it, and write it back to memory for events in the next cycle. However, concurrent read and write to memory in every cycle is costly. Instead of using a wide memory to serve all the transport events, the partitioning allows the data delivery and credit engines to have narrower memories to serve only the events that matter for their specific functionality, hence meeting timing constraints.

In this section, we present, in §3.1, how the engines coordinate to fairly and efficiently pick one of a few thousand flows every cycle for segment transmission while keeping the outgoing link utilized. Next, §3.2 and §3.3 describe fixed-function and programmable event processing modules in each engine, and how their design is inspired by patterns in Table 1. We present Tonic's solution for resolving conflicts when multiple events for the same flow are received in a cycle in §3.4, and its programming interface in §3.5.

3.1 Efficient Flow Scheduling

At any time, a flow can only transmit a data segment if it (1) has enough credit, and (2) has a new or lost segment to send. To be work conserving, Tonic must track the set

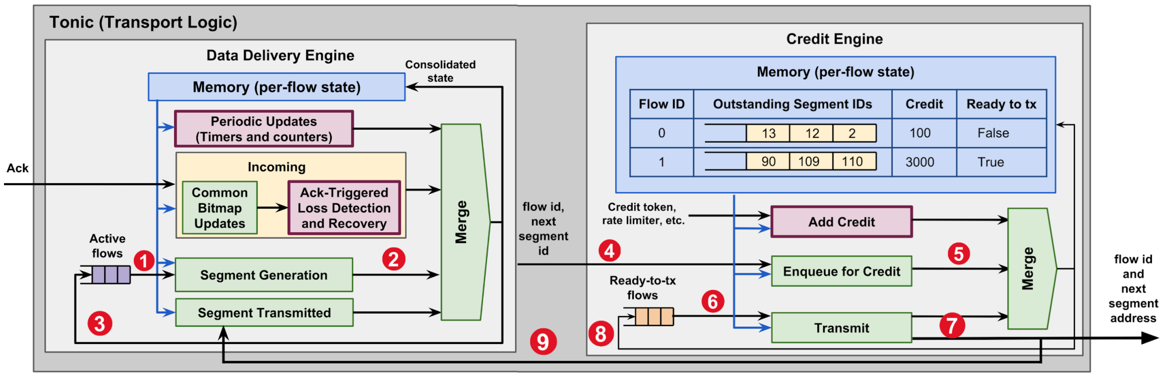


Figure 2: Tonic's architecture (dark red boxes (also with thick borders) are programmable, others are fixed)

of flows that are eligible for transmission (meet both of the above criteria) and only pick among those when selecting a flow for transmission each cycle. This is challenging to do efficiently. We have more than a thousand flows with their state partitioned across the two engines: Only the credit engine knows how much credit a flow has, and only the data delivery engine knows the status of a flow's segments and can generate the address of its next segment. We cannot check the state of all the flows every cycle across both engines to find the ones eligible for transmission in that cycle.

Instead, we decouple the *generation* of segment addresses from their *final transmission* to the DMA pipeline. We allow the data delivery engine to generate up to N segment addresses for a flow without necessarily having enough credit to send them out. In the credit engine, we keep a ring buffer of size N for each flow to store these outstanding segments addresses. When the flow has enough credit to send a segment, the credit engine dequeues and outputs a segment address from the buffer and signals the data delivery engine to decrement the number of outstanding segments for that flow.

This solves the problem of the partitioned state across the two engines. The data delivery engine does not need to keep track of the credit changes of flows for segment address generation. It only needs to be notified when a segment address is dequeued from the buffer. Moreover, the credit engine does not need to know the exact status of all flow's segments. If the flow's ring buffer is empty, that flow does not have segments to send. Otherwise, there are already segment addresses that can be output when the flow has enough credit.

Still, the data delivery engine cannot simply check the state of all the flows every cycle to determine those that can generate segments. Instead, we dynamically maintain the set of *active* flows in the data delivery engine, i.e., the flows that have at least one segment to generate and less than N outstanding segments (see red numbered circles in Figure 2). When a flow is created, it is added to the active set. Every cycle, one flow is selected and removed from the set for segment generation (Step 1). Once processed (Step 2), only if it has more segments to send and less than N outstanding segments, is it inserted back into the set (Step 3). Otherwise, it will be inserted in the set if, later on, the receipt of an ack or a signal from the credit engine "activates" the flow (Step 9). Moreover, the generated segment address is forwarded to the credit engine (Step 4) for insertion in the ring buffer (Step 5).

Similarly, the credit engine maintains the set of *ready-to-transmit* flows, i.e., the flows with one segment address or more in their ring buffers and enough credit to send at least one segment out. Every cycle, a flow is selected from the set (Step 6), one segment address from its ring buffer is transmitted (Step 7), its credit is decreased, and it is inserted back into the set if it has more segment addresses and credit for further transmission (Step 8). It also signals the data delivery engine about the transmission (Step 9) to decrement the number of outstanding segments for that flow.

To be fair when picking flows from the active (or ready-to-transmit) set, Tonic uses a FIFO to implement round-robin scheduling among flows in the set (see active list in [39]). The choice of round-robin scheduling is not fundamental; any other scheduler that meets our timing constraints can replace the FIFO to support other scheduling disciplines [40].

3.2 Flexible Segment Selection

With B bytes of credit, a flow can send $S = max(B, MSS)$ bytes, where MSS is the maximum segment size. In transport protocols, data delivery algorithms use acknowledgments to keep track of the status of each byte of data (e.g., delivered, lost, in-flight, and not transmitted), and use that to decide which contiguous S bytes of data to transmit next.

However, there are two main challenges in implementing data delivery algorithms in high-speed NICs. First, due to memory constraints, the NIC cannot store per-byte information. Second, with a few exceptions [8, 34], these algorithms are designed for software, where they could store and freely loop through large arrays of metadata to aggregate information. This computational flexibility has created significant diversity across these algorithms. Unfortunately, NIC hardware is much more constrained than software. Thus, we did not aim to support all data delivery algorithms. Instead, we looked for patterns that are common across a variety of algorithms while being amenable to hardware implementation.

3.2.1 Pre-Calculated Fixed Segment Boundaries

Data delivery algorithms could conceivably choose the next S bytes to send from anywhere in the data stream and produce segments with variable boundaries. However, since the NIC cannot maintain per-byte state, Tonic requires data to be partitioned into fixed-size segments (by a Kernel module or the driver, see §5) when the flow requests transmission of new data. This way, data delivery algorithms can use *per-segment* information to select the next segment.

Note that the fixed segment size can be configured for each flow based on its throughput and latency requirements. With message-based transport protocols (e.g., RoCEv2), having fixed segment boundaries fits naturally; the message length is known and the optimal segment size can be chosen from the beginning. For protocols with a byte-stream abstraction (e.g., TCP and NDP), the fixed segment size should be decided on the fly as data is added to the stream. It can be set to MSS (or larger if using TSO [18]) for high-bandwidth flows. For flows that generate small data segments and sporadically, the segment size can be set to a smaller value, depending on whether it is more desirable to consolidate multiple small segments into a larger one before notifying Tonic, or to transmit the small segment right away (§5). Regardless, to avoid storing per-byte state on the NIC, segment size should be decided outside of Tonic and changed infrequently.

3.2.2 Small Per-Segment State for a Limited Window

Independent of a flow's available credit, data delivery algorithms typically do not transmit a new segment if it is too far, i.e., more than K segments apart, from the first unacknowledged segment, to limit the state that the sender and receiver need to keep [2]. Still, in a 100 Gbps network with a 10μs RTT, K can get as large as $\sim$128 segments. Fortunately, we observe that storing the following per-segment state is enough for most data delivery algorithms: (1) Is the segment acknowledged (in presence of selective acknowledgments)? (2) If not, is it lost or still in flight? (3) If lost, is it already retransmitted (to avoid redundant retransmission)?

More specifically, we observe that, in the absence of explicit negative acknowledgments, data delivery algorithms accumulate evidence of loss for each segment from positive acknowledgments, e.g., duplicate cumulative (e.g., TCP NewReno [23]) or selective acks (e.g., IRN for RDMA and TCP SACK [16]). Once the accumulated evidence for a segment passes a threshold, the algorithm can declare it lost with high confidence. Typically, an evidence of loss for segment i is also an evidence of loss for every *unacknowledged* segment j with $j < i$. Thus, most of these algorithms can be rewritten to only keep track of the total evidence of loss for the first unacknowledged segment and incrementally compute the evidence for the rest as needed. Based on these observations (#1 and #2 in Table 1), we use a fixed set of bitmaps in Tonic's data delivery engine to track the status of a flow's segments and implement optimized fixed-function bitmap operations for updating them on transport events.

[2]In TCP-based protocols, K is the minimum of receive window and congestion window size. However, the limit imposed by K exists when transport protocols use other ways (e.g., rate) to limit a flow's transmission pace [8].

3.2.3 Concurrent Event Processing

For every flow, four main events can affect the generation of its next segment address. First, the receipt of an acknowledgment can either move the window forward and enable the flow to generate more segments, or signal segment loss and trigger retransmissions. Second, the absence of acknowledgments, i.e., a timeout, can also lead to more segments marked as lost and trigger retransmissions. Third, generation of a segment address increments the number of a flow's outstanding segments and can deactivate the flow if it goes above N. Fourth, segment address transmission (out of the credit engine) decrements the number of outstanding segments and can enable the flow to generate more segment addresses.

Tonic's data delivery engine has four modules to handle these four events (Figure 2). Every cycle, each module reads the state of the flow for which it received an event from the memory in the data delivery engine, processes the event, and updates the flow state accordingly. The flow state in the data delivery engine consists of a fixed set of variables to track the status of the current window of segments across events, as well as the user-defined variables used in the programmable components (Table 2). As an example of the fixed state variables, Tonic keeps a fixed set of bitmaps for each flow (observations in §3.2.2): The `acked` bitmap keeps track of selectively acknowledged segments, `marked-for-rtx` keeps track of lost segments that require retransmission, and `rtx-cnt` stores information about their previous retransmissions.

The following paragraphs briefly describe how each event-processing module affects a flow's state, and whether there are common patterns that we can exploit to implement all or parts of its functionality in a fixed-function manner.

Incoming. This module processes acknowledgments (and other incoming packets, see §3.3.3). Some updates to state variables in response to acknowledgments are similar across all data delivery algorithms and do not need to be programmable (e.g., updating window boundaries, and marking selectively acked segments in `acked` bitmap, see §3.2.2), whereas loss detection and recovery, which rely on acknowledgments as a signal, vary a lot across different algorithms and must be programmable by users (#4 in Table 1). Thus, the Incoming module is designed as a two-stage pipeline: a fixed-function stage for the common updates followed by a programmable stage for loss detection and recovery.

The benefit of this two-stage design is that the common updates mostly involve bitmaps and arrays (§3.2.2), which are implemented as ring buffers in hardware and costly to modify across their elements. For instance, in all data delivery algorithms, if an incoming packet acknowledges segment A cumulatively and segment S selectively, `wnd-start` is updated to `max(wnd-start, A)` and `acked[S]` to one, and the boundaries of all bitmaps and arrays are updated based on the new `wnd-start`. By moving these updates into a fixed function stage, we can (i) optimize them to meet Tonic's timing and memory constraints, and (ii) provide programmers with a dedicated stage, i.e., a separate cycle, to do loss detection and recovery. In this dedicated stage, programmers can use the updated state variables from the previous stage and the rest of the variables from memory to infer segment loss (and perform other user-defined computation discussed in §3.3.3).

Periodic Updates. The data delivery engine iterates over active flows, sending them one at a time to this module to check for retransmission timer expiration and perform other user-defined periodic updates (§3.3.3). Thus, with its 10 ns clock cycle, Tonic can cover each flow within a few microseconds of the expiry of its retransmission timer. This module must be programmable as a retransmission timeout is a signal for detecting loss (#4 in Table 1). Similar to the programmable stage of the Incoming module, the programmers can use per-flow state variables to infer segment loss.

Segment Generation. Given an active flow and its variables, this module generates the next segment's address and forwards it to the credit engine. Tonic can implement segment address generation as a fixed function module based on the following observation (#3 in Table 1): Although different reliable data delivery algorithms have different ways of inferring segment loss, once a lost segment is detected, it is only logical to retransmit it before sending anything new. Thus, the procedure for selecting the next segment is the same irrespective of the data delivery algorithm, and is implemented as a fixed-function module in Tonic. Thus, this module prioritizes retransmission of lost segments in `marked-for-rtx` over sending the next new segment, i.e., `highest_sent+1` and also increments the number of outstanding segments.

Segment Transmitted. This module is fixed function and is triggered when a segment address is transmitted out of the credit engine. It decrements the number of outstanding segments of the corresponding flow. If the flow was deactivated due to a full ring buffer, it is inserted into the active set again.

3.3 Flexible Credit Management

Transport protocols use congestion-control algorithms to avoid overloading the network by controlling the pace of a flow's transmission. These algorithms consist of a control loop that estimates the network capacity by monitoring the stream of incoming control packets (e.g., acknowledgments and congestion notification packets (CNPs)) and sets parameters that limit outgoing data packets. While the control loop is different in many algorithms, the credit calculation based on parameters is not. Tonic has efficient fixed-function modules for credit calculation (§3.3.1 and §3.3.2) and relegates parameter adjustment to programmable modules (§3.3.3).

State Variable	Description
acked	selectively acknowledged segments (bitmap)
marked-for-rtx	lost segments marked for retransmission (bitmap)
rtx-cnt	number of retransmissions of a segment (bitmap)
wnd-start	the address of the first segment in the window
wnd-size	size of the window ($min(W, rcved\_window)$)
highest-sent	the highest segment transmitted so far
total-sent	Total number of segments transmitted so far
is-idle	does the flow have segments to send?
outstanding-cnt	# of outstanding segments
rtx-timer	when will the rtx timer expire?
user-context	user-defined variables for programmable modules

Table 2: Per-flow state variables in the data delivery engine

3.3.1 Common Credit-Calculation Patterns

Congestion control algorithms have a broad range of ways to estimate network capacity. However, they enforce limits on data transmission in three main ways (#5 in Table 1):
Congestion window. The control loop limits a flow to at most W bytes in flight from the first unacknowledged byte. Thus, if byte i is the first unacknowledged byte, the flow cannot send bytes beyond $i+W$. Keeping track of in-flight segments to enforce a congestion window can get complicated, e.g., in the presence of selective acknowledgments, and is implemented in the fixed-function stage of the incoming module in the data delivery engine.
Rate. The control loop limits the flow's average rate (R) and maximum burst size (D). Thus, if a flow had credit c_0 at the time t_0 of the last transmission, then the credit at time t will be $min(R*(t-t_0)+c_0, D)$. As we show in §4, implementing precise per-flow rate limiters under our strict timing and memory constraints is challenging and has an optimized fixed-function implementation in Tonic.
Grant tokens. Instead of estimating network capacity, the control loop receives tokens from the receiver and adds them to the flow's credit. Thus, the credit of a flow is the total tokens received minus the number of transmitted bytes, and the credit calculation logic consists of a simple addition.

Since these are used by most congestion control algorithms[3], we optimize their implementation to meet Tonic's timing and memory constraints. Congestion window calculations are mostly affected by acks. Thus, calculation and enforcement of congestion window happen in the data delivery engine. For the other two credit calculation schemes, the credit engine processes credit-related event, and Tonic users can simply pick which scheme to use in the credit engine.

3.3.2 Event Processing for Credit Calculation

Conceptually, three main events can trigger credit calculation for a flow, and the credit engine has different modules to concurrently process them every cycle (Figure 2). First, when a segment address is received from the data delivery engine and is the only one in the flow's ring buffer, the flow could now qualify for transmission or remain idle based on

[3] Tonic's credit engine has a modular event-based design (§3.3.2), making it amenable for extension to future credit calculation schemes.

its credit (the Enqueue module). Second, when a flow transmits a segment address, its credit must be decreased and we should determine whether it is qualified for further transmission based on its updated credit and the occupancy of its ring buffer (the Transmit module). Third are events that can add credit to the flow (e.g., from grant tokens and leaky bucket rate limiters), which is where the main difference lies between rate-based and token-based credit calculation.

When using grant tokens, the credit engine needs two dedicated modules to add credit to a flow: one to process incoming grant tokens from the receiver, and one to add credit for retransmissions on timeouts. When using rate, the credit engine does not need any extra modules for adding credit since a flow with rate R bytes-per-cycle implicitly gains R bytes of credit every cycle and, therefore, we can compute in advance when it will be qualified for transmission.

Suppose in cycle T_0, the Transmit module transmits a segment from flow f, and is determining whether the flow is qualified for further transmission. Suppose that f has more segments in the ring buffer but lacks L bytes of credit. The Transmit module can compute when it will have sufficient credit as $T = \frac{L}{R}$ and set up a timer for T cycles. When the timer expires, f definitely has enough credit for at least one segment, so it can be directly inserted into `ready-to-tx`. When f reaches the head of `ready-to-tx` and is processed by the Transmit module again in cycle T_1, the Transmit module can increase f's credit by $(T_1 - T_0) * R - S$, where S is the size of the segment that is transmitted at time T_1 [4]. Note that when using rate, the credit engine must perform division and maintain per-flow timers. We will discuss the hardware implementation of these operations in §4.

3.3.3 Flexible Parameter Adjustment

Congestion control algorithms often have a control loop that continuously monitors the network and adjusts credit calculation parameters, i.e., rate or window size, based on estimated network capacity. Parameter adjustment is either triggered by incoming packets (e.g., acknowledgments and their signals such as ECN or delay in TCP variants and Timely, and congestion notification packets (CNPs) in DCQCN) or periodic timers and counters (timeouts in TCP variants and byte counter and various timers in DCQCN), and in some cases is inspired by segment losses as well (window adjustment after duplicate acknowledgments in TCP).

Corresponding to these triggers, for specifying parameter adjustment logic, Tonic's users can use the programmable stage of the "Incoming" module, which sees all incoming packets, and the "Periodic Updates" module for timers and counters. Both modules are in the data delivery engine and have access to segment status information, in case segment status (e.g., drops) is needed for parameter adjustment. The updated parameters are forwarded to the credit engine.

[4]Similarly, the Enqueue module can set up the timer when it receives the first segment of the queue and the flow lacks credit for its transmission.

As we show in §6.1.1, we have implemented several congestion control algorithms in Tonic and their parameter adjustment calculations have finished within our 10 ns clock cycle. Those with integer arithmetic operations did not need any modifications. For those with floating-point operations, such as DCQCN, we approximated the operations to a certain decimal point using integer operations. If an algorithm requires high-precision and complicated floating-point operations for parameter adjustment that cannot be implemented within one clock cycle [19], the computation can be relegated to a floating-point arithmetic module outside of Tonic. This module can perform the computation asynchronously and store the output in a separate memory, which merges into Tonic through the "Periodic Updates" module.

3.4 Handling Conflicting Events

Tonic strives to process events concurrently in order to be responsive to events. Thus, if a flow receives more than one event in the same cycle, it allows the event processing modules to process the events and update the flow's state variables, and reconciles the state before writing it back into memory (the Merge modules in Figure 2).

Since acknowledgments and retransmission timeouts are, by definition, mutually exclusive, Tonic discards the timeout if it is received in the same cycle as an acknowledgment for the same flow. This significantly simplifies the merge logic because several variables (window size and retransmission timer period) are *only* modified by these two events and, therefore, are never updated concurrently. We can resolve concurrent updates for the remaining variables with simple, predefined merge logic. For example, Segment Generation increments the number of outstanding segments, whereas Segment Transmitted decrements it; if both events affect the same flow at the same time, the number does not change. User-defined variables are updated in either the Incoming or the Periodic Updates module, and we rely on the programmer to specify which updated variables should be prioritized if both updates happen in the same cycle.

3.5 Tonic's Programming Interface

To implement a new transport logic in Tonic, programmers only need to specify (i) which of the three credit management schemes to use, (ii) the loss detection and recovery logic in response to acknowledgments and timeouts, and (iii) congestion-control parameter adjustment in response to incoming packets or periodic timers and counters. The first one is used to pick the right modules for the credit engine, and the last two are inserted into the corresponding programmable stages of the data delivery engine (Figure 2).

To specify the logic for the programmable stage of the Incoming module, programmers need to write a function that receives the incoming packet (ack or other control signals), the number of newly acknowledged segments, the `acked` bitmap updated with the information in the ack, the old and new value of `wnd-start` (in case the window moves forward due to a new cumulative ack), and the rest of the flow's state variables (Table 2) as input. In the output, they can mark a range of segments for retransmission in `marked-for-rtx`, update congestion-control parameters such as window size and rate, and reset the retransmission timer. The programming interface of the Periodic Updates module is similar.

In specifying these functions, programmers can use integer arithmetic operations, e.g., addition, subtraction, multiplication, and division with small-width operands, conditionals, and a limited set of read-only bitmap operations, e.g., index lookup, and finding the first set bit in the updated `acked` bitmap (see appendix F for an example program). Note that a dedicated fixed-function stage in the data delivery engine performs the costly common bitmap updates on receipt of acks (§3.2.3). We show, in §6.1.1, that a wide range of transport protocols can be implemented using this interface and give examples of ones that cannot.

4 Hardware Implementation

In this section, we describe the hardware design of the Tonic components that were the most challenging to implement under Tonic's tight timing and memory constraints.

High-Precision Per-Flow Rate Limiting. A flow with rate R bytes per cycle and L bytes to send will have sufficient credit for transmission in $T = \lceil \frac{L}{R} \rceil$ cycles. Tonic needs to do this computation in the credit engine but must represent R as an integer since it cannot afford to do floating-point division. This creates a trade-off between the rate-limiting precision and the range of rates Tonic can support. If R is in bytes per cycle, we cannot support rates below one byte per cycle or $\sim$1 Gbps. If we represent R in bytes per thousand cycles, we can support rates as low as 1 Mbps. However, $T = \lceil \frac{L}{R} \rceil$ determines how many thousand cycles from now the flow qualifies for transmission which results in lower rate conformance and precision for higher-bandwidth flows. To support a wide range of rates without sacrificing precision, Tonic keeps multiple representations of the flow's rate at different levels of precision and picks the most precise representation for computing T at any moment (details in Appendix B).

Efficient Bitmap Operations. Tonic uses bitmaps as large as 128 bits to track the status of segments for each flow. Bitmaps are implemented as ring buffers. The head pointer corresponds to the first unacked segment and moves forward around the buffer with new acks. To efficiently implement operations whose output depends on the values of *all* the bits in the bitmap, we must divide the buffer into smaller parts in multiple layers, process them in parallel, and join the results. One such operation, frequently used in Tonic, is finding the first set bit after the head. The moving head of the ring buffer complicates the implementation of this operation since keeping track of the head in each layer requires extra processing, making it difficult to compute within our 10 ns target. Instead, Tonic uses a light-weight pre-processing on the input ring buffer to avoid head index computation in the layers altogether (details in Appendix C).

Concurrent Memory Access. Every cycle, five modules in the data delivery engine, including both stages of the Incoming module, concurrently access its memory (§3.2.3). However, FPGAs only have dual-ported block RAMs, with each port capable of either read or write every cycle. Building memories with more concurrent reads and writes requires keeping multiple copies of data in separate memory "banks" and keeping track of the bank with the most recent data for each address[5] [26]. To avoid supporting *five* concurrent reads and writes, we manage to partition per-flow state variables into two groups, each processed by at most four events. Thus, Tonic can use two memories with four read and write ports instead of a single one with five, to provide concurrent access for all processing modules at the same time.

5 Integrating Tonic into the Transport Layer

Tonic's transport logic is intentionally decoupled from the specific implementation of other transport functionality such as connection management, application-level API, and buffer management. This section provides an example of how Tonic can interface with the Linux kernel to learn about new connections, requests for data transmission, and connection termination [6]. After creating the socket, applications use various system calls for connection management and data transfer. As Tonic mainly focuses on the sender sider of the transport logic, we only discuss the system calls and modifications relevant to the sender side of the transport layer.

Connection Management. `connect()` on the client initiates a connection, `listen()` and `accept()` on the server listen for and accept connections, and `close()` terminate connections. As connection management happens outside of Tonic, the kernel implementation of these system calls stays untouched. However, once the connection is established, the kernel maps it to a unique *flow id* in $[0, N)$, where N is the maximum number of flows supported by Tonic, and notifies Tonic through the NIC driver about the new connection.

Specifically, from the connection's Transmission Control Block (TCB) in the kernel, the IP addresses and ports of the communication endpoints are sent to Tonic alongside the flow id and the fixed segment size chosen for the connection. The kernel only needs to track the TCB fields used for connection management (e.g., IP addresses, ports, and TCP FSM), pointers to data buffers, and receiver-related fields. Fields used for data transfer on the sender, i.e., `snd.nxt`, `snd.una`, and `snd.wnd`, are stored in and handled by Tonic. Finally, after a call to `close()`, the kernel uses the connec-

[5] This overhead is specific to FPGAs, and can potentially be eliminated if the memory is designed as an ASIC.

[6] See appendix A for how Tonic can be used with RDMA.

tion's flow id to notify Tonic of its termination.

Data Transfer. `send()` adds data to the connection's socket buffer, which stores its outstanding data waiting for delivery. Tonic keeps a few bits of per-segment state for outstanding data and performs all transport logic computation in terms of segments. As such, data should be partitioned into equal-sized segments before Tonic can start its transmission (§3.2). Thus, modifications to `send()` mainly involve determining segment boundaries for the data in the socket buffer based on the connection's configured segment size and deciding when to notify Tonic of the new segments. Specifically, the kernel keeps an extra pointer for each connection's socket buffer, in addition to its `head` and `tail`, called `tonic-tail`. It points to the last segment of which Tonic has been notified. `head` and updates to `tonic-tail` are sent to Tonic to use when generating the next segment's address to fetch from memory.

Starting with an empty socket buffer, when the application calls `send()`, data is copied to the socket buffer, and `tail` is updated accordingly. Assuming the connection's configured segment size is C, the data is then partitioned into C-sized segments. Suppose the data is partitioned into S segments and $B < C$ remaining bytes. The kernel then updates `tonic-tail` to point to the end of the last C-sized segment, i.e., `head + C * S`, and notifies Tonic of the update to `tonic-tail`. The extra B bytes remain unknown to Tonic for a configurable time T, in case the application calls `send` to provide more data. In that case, the data are added to the socket buffer, data between `tonic-tail` and `tail` are similarly partitioned, `tonic-tail` is updated accordingly, and Tonic is notified of new data segments.

If there is not enough data for a C-sized segment after time T, the kernel needs to notify Tonic of the "sub-segment" (a segment smaller than C) and its size, and update `tonic-tail` accordingly. Note that Tonic requires all segments, except for the last one in a burst, to be of equal size, as all computations, including window updates, are in terms of segments. Thus, after creating a "sub-segment", if there is more data from the application, Tonic can only start its transmission when it is done transferring its current segments. Tonic notifies the kernel once it successfully delivers the final "sub-segment", at which point, `head` and `tonic-tail` will be equal, and the kernel continues partitioning the remaining data in the socket buffer and updating Tonic as before. Note that Tonic can periodically, with a configurable frequency, forward acknowledgments to the kernel to move `head` forward and free up space for new data in the socket buffer.

C and T can be configured for each flow based on its latency and throughput characteristics. For high-bandwidth flows, C can be set to MSS (or larger, if using TSO). For flows that sporadically generate small segments, setting C and T is not as straightforward since segments cannot be consolidated within Tonic. We discuss the trade-offs in deciding these parameters in detail in appendix D.

Other Considerations. As we show in §6, Tonic's current design supports 2048 concurrent flows, matching the working sets observed in data centers [15,37] and other hardware offloads in the literature [20]. If a host has more open connections than Tonic can support, the kernel can offload data transfer for flows to Tonic on a first-come first-serve basis, or have users set a flag when creating the socket and fall back to software once Tonic runs out of resources for new flows. Alternatively, modern FPGA-based NICs have a large DRAM directly attached to the FPGA [20]. The DRAM can potentially be used to store the state of more connections, and swap them back and forth into Tonic's memory as they activate and need to transmit data. Moreover, to provide visibility into the performance of hardware transport logic, Tonic can provide an interface for kernel to periodically pull transport statistics from the NIC.

Other Transport Layers. The above design is an example of how Tonic can be integrated into a commonly-used transport layer. However, TCP, sockets, and bytestreams are not suitable for all applications. In fact, several datacenter applications with high-bandwidth low-latency flows are starting to use RDMA and its message-based API instead [5,9,22,35]. Tonic can be integrated into RDMA-based transport as well, which we discuss in appendix A.

6 Evaluation

To evaluate Tonic, we implement a prototype in Verilog (∼8K lines of code) and a cycle-accurate hardware simulator in C++ (∼2K lines of code) [11]. The simulator is integrated with NS3 network simulator [4] for end-to-end experiments.

To implement a transport protocol on Tonic's Verilog prototype, programmers only need to provide three Verilog files: (i) `incoming.v`, describing the loss detection and recovery logic and how to change credit management parameters (i.e., rate or window) in response to incoming packets; this code is inserted into the second stage of the Incoming pipeline in the data delivery engine, (ii) `periodic_updates.v`, describing the loss detection and recovery logic in response to timeouts and how to change credit management parameters (i.e., rate or window) in response to periodic timers and counters; this code is inserted into the Periodic Updates module in the data delivery engine, and (iii) `user_configs.vh`, specifying which of the three credit calculation schemes to use and the initial values of user-defined state variables and other parameters, such as initial window size, rate, and credit.

We evaluate the following two aspects of Tonic:

Hardware Design (§6.1). We use Tonic's Verilog prototype to evaluate its hardware architecture for *programmability* and *scalability*. Can Tonic support a wide range of transport protocols? Does it reduce the development effort of implementing transport protocols in the NIC? Can Tonic support complex user-defined logic with several variables? How many per-flow segments and concurrent flows can it support?

End-to-End Behavior (§6.2). We use Tonic's cycle-accurate simulator and NS3 to compare Tonic's end-to-end behavior with that of hard-coded implementations of two protocols: New Reno [23] and RoCEv2 with DCQCN [43], both for a single flow and for multiple flows sharing a bottleneck link.

6.1 Hardware Design

There are two main metrics for evaluating the efficiency of a hardware design: (i) **Resource Utilization.** FPGAs consist of primitive blocks, which can be configured and connected to implement a Verilog program: *look-up tables (LUTs)* are the main reconfigurable logic blocks, and *block RAMs (BRAMs)* are used to implement memory. (ii) **Timing.** At the beginning of each cycle, each module's input is written to a set of input registers. The module must process the input and prepare the result for the output registers before the next cycle begins. Tonic must *meet timing* at 100 MHz to transmit a segment address every 10 ns. That is, to achieve 100 Gbps, the processing delay of every path from input to output registers in every module must stay within 10 ns.

We use these two metrics to evaluate Tonic's programmability and scalability. These metrics are highly dependent on the specific target used for synthesis. We use the Kintex Ultrascale+ XCKU15P FPGA as our target because this FPGA, and others with similar capabilities, are included as bump-in-the-wire entities in today's commercial programmable NICs [2, 3]. This is a conservative choice, as these NICs are designed for 10-40 Gbps Ethernet. A 100 Gbps NIC could potentially have a more powerful FPGA. Moreover, we synthesize *all* of Tonic's components onto the FPGA to evaluate it as a standalone prototype. However, given the well-defined interfaces between the fixed-function and programmable modules, it is conceivable to implement the fixed-function components as an ASIC for more efficiency. Unless stated otherwise, we set the maximum number of concurrent flows to 1024 and the maximum window size to 128 segments in all of our experiments [7].

6.1.1 Hardware Programmability

We have implemented the sender's transport logic of six protocols in Tonic as representatives of various types of segment selection and credit calculation algorithms in the literature. Table 3 summarizes their resource utilization for both fixed-function and user-defined modules, and the lines of code and bytes of user-defined state used to implement them. While we use the same set of per-flow state variables (Table 2) for all protocols, not all of them use all the variables in processing transport events. For instance, bitmaps are only used by protocols with selective acks. Thus, it is possible to reduce the resource utilization even more with some pre-processing to remove the irrelevant variables and computation from the Verilog design.

[7]A 100 Gbps flow with 1500B back-to-back packets over 15-μs RTT, typical in data centers, has at most 128 in-flight segments.

	User-Defined Logic		**Credit Type**	**Look up Tables (LUTs)**				**BRAMs**	
				User-Defined		**Fixed**			
	LoC	state(B)		total(K)	%	total(K)	%	total	%
Reno	48	8	wnd	2.4	0.5	109.4	20.9	195	20
NewReno	74	13	wnd	2.6	0.5	112.5	21.5	211	21
SACK	193	19	wnd	3.3	0.6	112.1	21.4	219	22
NDP	20	1	token	3.0	0.6	143.6	29.0	300	30
RoCE w/ DCQCN	63	30	rate	0.9	0.2	185.2	35.2	251	26
IRN	54	14	rate	2.9	0.6	177.4	33.9	219	22

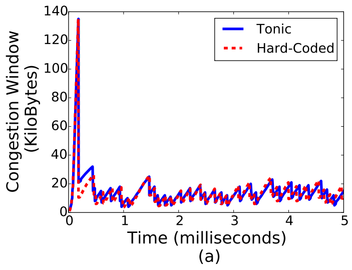
Table 3: Resource utilization of transport protocols in Tonic.

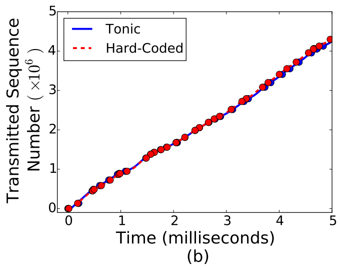
Reno [13] and New Reno [23] represent TCP variants that use only cumulative acks for reliable delivery and congestion window for credit management. Reno can only recover from one loss within the window using fast retransmit, whereas New Reno uses partial acknowledgments to recover more efficiently from multiple losses in the same window. SACK, inspired by RFC 6675 [16], represents TCP variants that use selective acks. Our implementation has one SACK block per ack but can be extended to more. NDP [24] represents receiver-driven protocols, recently proposed for low-latency data-center networks [21, 36]. It uses explicit NACKs and timeouts for loss detection and grant tokens for congestion control. RoCEv2 w/ DCQCN [43] is a widely-used transport for RDMA over Ethernet, and IRN [34] is a recent hardware-based protocol for improving RoCE's simple data delivery algorithm. Both use rate limiters for credit management.

Note that, as described in §3.2, not all data delivery algorithms are feasible for hardware implementation as is. For instance, due to memory constraints on the NIC, it is not possible to keep timestamps for *every* packet, new and retransmissions, on the NIC. As a result, transport protocols which rely heavily on per-packet timestamps, e.g., QUIC [27], need to be modified to work with fewer timestamps, i.e., for a subset of in-flight segments, to be offloaded to hardware.

Takeways. There are three key takeaways from these results:

- *Tonic supports a variety of transport protocols.*
- *Tonic enables programmers to implement new transport logic with modest development effort.* Using Tonic, each of the above protocols is implemented in less than 200 lines of Verilog code, with the user-defined logic consuming less than 0.6% of the FPGA's LUTs. In contrast, Tonic's fixed-function modules, which are reused across these protocols, are implemented in ∼8K lines of code and consume ∼60 times more LUTs.
- *Different credit management schemes have different overheads.* For transport protocols that use congestion window, window calculations overlap with and therefore are implemented in the data delivery engine (§3.3.1). Thus, their credit engine utilizes fewer resources than others. Rate limiting requires more per-flow state and more complicated operations (§4) than enforcing receiver-generated grant tokens but needs fewer memory ports for concurrent reads and writes (§3.3.2), overall leading to lower BRAM and higher LUT utilization for rate limiting.





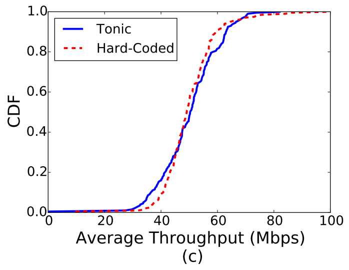


Figure 3: NewReno's Tonic vs hard-coded implementation in NS3 (10G line-rate): a) Congestion window updates (single flow, random drops), b) Transmitted sequence numbers with retransmission in large dots (single flow, random drops), and c) CDF of average throughput of multiple flows sharing a bottleneck link over 5 seconds (200 flows from 2 hosts to one receiver)

6.1.2 Hardware Scalability

We evaluate Tonic's scalability by examining how sources of variability in its architecture (programmable modules and various parameters) affect memory utilization and timing.

User-defined logic in programmable modules can have arbitrarily-long chains of dependent operations, potentially causing timing violations. We generate 70 random programs for `incoming.v` (the programmable stage of Incoming module in data delivery engine) with different numbers of arithmetic, logical, and bitmap operations, and analyze how long the chain of dependent operations gets without violating timing at 10ns. These programs use up to 125B of state and have a maximum dependency of 65 *logic levels* (respectively six and two times more than the benchmark protocols in Table 3). Each logic level represents one of several primitive logic blocks (LUT, MUX, DSP, etc.) chained together to implement a path in a Verilog program.

We plug these programs into Tonic, synthesize them, and analyze the relationship between the number of logic levels and latency of the max-delay path (Table 4). Programs with up to 32 logic levels consistently meet timing, while those with more than 43 logic levels do not. Between 32 and 42 logic levels, the latency of the max-delay path is around 10 ns. Depending on the mix of primitives on the max-delay path and their latencies, programs in that region can potentially meet timing. Our benchmark protocols have 13 to 29 logic levels on their max-delay path and all meet timing. Thus, Tonic not only supports our benchmark protocols, but also has room to support future more sophisticated protocols.

User-defined state variables increase the memory width affecting BRAM utilization. We add extra variables to SACK, IRN, and NDP to see how wide memories can get without violating timing and running out of BRAMs, repeating the experiment for each of the three credit management schemes as they have different memory footprints (Table 4). Tonic can support 448B of user-defined state with congestion window for credit management, 340B with rate, and 256B with grant tokens (Protocols in Table 3 use less than 30B).

Maximum window size determines the size of per-flow bitmaps stored in the data delivery engine to keep track of the status of a flow's segments, therefore affecting memory utilization, and the complexity of bitmap operations, hence timing. Tonic can support bitmaps as large as 256 bits (i.e., tracking 256 segments), with which we can support a single 100Gbps flow in a network with up to 30µs RTT (Table 4).

	Metric	Results	
Complexity of User-Defined Logic	logic levels	(0 ,31]	meets timing
		(31,42]	depends on operations
		(42,65]	violates timing
User-Defined State	bytes	256	grant token
		340	rate
		448	congestion window
Window Size	segments	256	
Concurrent Flows	count	2048	

Table 4: Summary of Tonic's scalability results.

Maximum number of concurrent flows determines memory depth and the size of FIFOs used for flow scheduling (§3.1). Thus, it affects both memory utilization and the queue operations, hence timing. Tonic can scale to 2048 concurrent flows in hardware (Table 4) which matches the size of the active flow set observed in data centers [15, 37] and other hardware offloads in the literature [20].

Takeways. Tonic has additional room to support future protocols that are more sophisticated with more user-defined variables than our benchmark protocols. It can track 256 segments per flow and support 2048 concurrent flows. With a more powerful FPGA with more BRAMs, Tonic can potentially support even larger windows and more flows.

6.2 End-to-End Behavior

To examine Tonic's end-to-end behavior and verify the fidelity of Tonic-based implementation of the transport logic in different protocols, we have developed a cycle-accurate hardware simulator for Tonic in C++. We use this simulator with NS3 to show that Tonic-based implementation of NewReno and RoCEv2 w/ DCQCN senders match their hard-coded NS3 implementation. Note that the goal of these simulations is to analyze and verify Tonic's end-to-end behavior. Tonic's ability to support 100Gbps line rate has already been demonstrated in §6.1 using hardware synthesis. Thus, in our simulations, we use 10Gbps and 40Gbps as line rate merely to make hardware simulations with multiple flows over seconds computationally tractable.

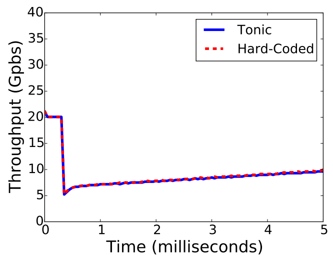


Figure 4: RoCEv2 w/ DCQCN in Tonic vs hard-coded in NS3 (40G line rate, one of two flows on a bottleneck link).

6.2.1 TCP New Reno

We implement TCP New Reno in Tonic based on RFC 6582, and use NS3's native network stack for its hard-coded implementation. Our Tonic-based implementation works with the *unmodified* native TCP receiver in NS3. In all simulations, hosts are connected via 10Gbps links to one switch, RTT is $10\mu s$, the buffer is 5.5MB, the minimum retransmission timeout is 200ms (Linux default), segments are 1000B large, and delayed acks are enabled on the receiver.

Single Flow. We start a single flow from one host to another, and randomly drop packets on the receiver's NIC. Figure 3(a) and 3(b) show the updates to the congestion window and transmitted sequence numbers (retransmissions are marked with large dots), respectively. Tonic's behavior in both cases closely matches the hard-coded implementation. The slight differences stem from the fact that in NS3's network stack, all the computation happens in the same virtual time step while in Tonic every event (incoming packets , segment address generation, etc.) is processed over a 100ns cycle (increased from 10ns to match the 10G line rate).

Multiple Flows. Two senders each start 100 flows to a single receiver, so 200 flows share a single bottleneck link for 5 seconds. The CDF of average throughput across the 200 flows for the Tonic-based implementation closely matches that of the hard-coded implementation (Figure 3(c)). We observe similarly matching distributions for number of retransmissions. When analyzing the flows' throughput in millisecond-long epochs, we notice larger variations in the hard-coded implementation than Tonic since Tonic, as opposed to NS3's stack, performs per-packet round robin scheduling across flows on the same host.

6.2.2 RoCEv2 with DCQCN

We implement RoCE w/ DCQCN [43] in Tonic, and use the authors' NS3 implementation from [44] for the hard-coded implementation. Our Tonic-based implementation works with the *unmodified* hard-coded RoCE receiver. In all simulations, hosts are connected via 40Gbps links to the same switch, RTT is $4\mu s$, segments are 1000B large, and we use the default DCQCN parameters from [44].

Single Flow. DCQCN is a rate-based algorithm which uses CNPs and periodic timers and counters for congestion control as opposed to packet loss in TCP. Thus, to observe rate updates for a single flow, we run two flows from two hosts to the same receiver for one second to create congestion and track the throughput changes of one as they both converge to the same rate. Tonic's behavior closely matches the hard-coded implementation (Figure 4). We also ran a single DCQCN flow at 100Gbps with 128B back-to-back packets and confirmed that Tonic can saturate the 100Gbps link.

Multiple Flows. Two senders each start 100 flows to a single receiver, so 200 flows share a single bottleneck link for one second. Both Tonic and the hard-coded implementation do per-packet round robin scheduling among the flows on the same host. As a result, all flows in both cases end up with an average throughput of 203 ± 0.2Mbps. Moreover, we observe a matching distribution of CNPs in both cases.

7 Related Work

Tonic is the first programmable architecture for transport logic in hardware able to support 100 Gbps. In this section, we review the most closely related prior work.

Commercial hardware network stacks. Some NICs have hardware network stacks with hard-wired transport protocols [8, 10]. However, they only implement two protocols, either RoCE [8] or a vendor-selected TCP variant, and can only be modified by their vendor. Tonic enables programmers to implement a variety of transport protocols in hardware with modest effort. In the absence of a publicly-available detailed description of these NICs' architecture, we could not compare our design decisions with theirs.

Non-commercial hardware transport protocols. Recent work explores hardware transport protocols that run at high speed with low memory footprint [30, 31, 34]. Tonic facilitates innovation in this area by enabling researchers to implement new protocols with modest development effort.

Accelerating network functionality. Several academic and industrial projects offload end-host virtual switching and network functions to FPGAs, processing a stream of already-generated packets [14, 20, 28, 29, 41]. Tonic, on the other hand, implements the transport logic in the NIC by keeping track of potentially a few hundred segments at a time to generate packets at line rate while running user-defined transport logic to ensure efficient and reliable delivery.

Acknowledgments

We thank Aditya Akella, our shepherd, the anonymous reviewers of NSDI'20, Hari Balakrishnan, Gordon Bebner, and Radhika Mittal for their helpful feedback. We thank Behnaz Arzani, Xiaoqi Chen, Rob Harrison, Suriya Kodeswaran, Shir Landau-Feibish, Srinivas Narayana, Praveen Tammana, and Ross Teixeira for feedback on earlier drafts of this work. This work is supported by DARPA under Dispersed Computing HR0011-17-C-0047, NSF Grants No. CCF-1822949 and CCF-1453112, and DARPA No. FA8650-18-2-7846.

References

[1] F-Stack. http://www.f-stack.org/. Accessed: August 2019.

[2] Innova Flex 4 Lx EN Adapter Card. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova_Flex4_Lx_EN.pdf. Accessed: August 2019.

[3] Mellanox Innova 2 Flex Open Programmable SmartNIC. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Innova-2_Flex.pdf. Accessed: August 2019.

[4] NS3 Network Simulator. https://www.nsnam.org/. Accessed: August 2019.

[5] NVMe over Fabric. https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf. Accessed: August 2019.

[6] OpenOnload. https://www.openonload.org/. Accessed: August 2019.

[7] RDMA - iWARP. https://www.chelsio.com/nic/rdma-iwarp/. Accessed: August 2019.

[8] RDMA and RoCE for Ethernet Network Efficiency Performance. http://www.mellanox.com/page/products_dyn?product_family=79&mtag=roce. Accessed: August 2019.

[9] RoCE Accelerates Data Center Performance, Cost Efficiency, and Scalability. http://www.roceinitiative.org/wp-content/uploads/2017/01/RoCE-Accelerates-DC-performance_Final.pdf. Accessed: August 2019.

[10] TCP Offload Engine (TOE). https://www.chelsio.com/nic/tcp-offload-engine/. Accessed: August 2019.

[11] Tonic Github Repository. https://github.com/minmit/tonic.

[12] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).

[13] Allman, M., Paxson, V., and Blanton, E. TCP Congestion Control. RFC 5681, 2009.

[14] Arashloo, M. T., Ghobadi, M., Rexford, J., and Walker, D. HotCocoa: Hardware congestion control abstractions. In *HotNets* (2017).

[15] Benson, T., Akella, A., and Maltz, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *IMC* (2010).

[16] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Nishida, Y. A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP. RFC 6675, 2012.

[17] Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., and Jacobson, V. BBR: Congestion-Based Congestion Control. *ACM Queue* (2016).

[18] Connery, G. W., Sherer, W. P., Jaszewski, G., and Binder, J. S. Offload of TCP Segmentation to a Smart Adapter, 1999. US Patent 5,937,169.

[19] Dong, M., Li, Q., Zarchy, D., Godfrey, P. B., and Schapira, M. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI* (2015).

[20] Firestone, D., et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI* (2018).

[21] Gao, P. X., Narayan, A., Kumar, G., Agarwal, R., Ratnasamy, S., and Shenker, S. pHost: Distributed Near-Optimal Datacenter Transport Over Commodity Network Fabric. In *CoNEXT* (2015).

[22] Guo, C., Wu, H., Deng, Z., Soni, G., Ye, J., Padhye, J., and Lipshteyn, M. RDMA over Commodity Ethernet at Scale. In *SIGCOMM* (2016).

[23] Handerson, T., Floyd, S., Gurtov, A., and Nishida, Y. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, 1999.

[24] Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., and Wójcik, M. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM* (2017).

[25] Jeong, E., Woo, S., Jamshed, M. A., Jeong, H., Ihm, S., Han, D., and Park, K. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI* (2014).

[26] LaForest, C. E., and Steffan, J. G. Efficient Multi-Ported Memories for FPGAs. In *FPGA* (2010).

[27] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., et al. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM* (2017).

[28] Lavasani, M., Dennison, L., and Chiou, D. Compiling High Throughput Network Processors. In *FPGA* (2012).

[29] Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. Clicknp: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM* (2016).

[30] Lu, Y., Chen, G., Li, B., Tan, K., Xiong, Y., Cheng, P., Zhang, J., Chen, E., and Moscibroda, T. Multi-Path Transport for RDMA in Datacenters. In *NSDI* (2018).

[31] Lu, Y., Chen, G., Ruan, Z., Xiao, W., Li, B., Zhang, J., Xiong, Y., Cheng, P., and Chen, E. Memory Efficient Loss Recovery for Hardware-Based Transport in Datacenter. In *Asia-Pacific Workshop on Networking* (2017).

[32] Marinos, I., Watson, R. N., and Handley, M. Network Stack Specialization for Performance. In *SIGCOMM* (2014).

[33] Mathis, M., and Mahdavi, J. Forward Acknowledgement: Refining TCP Congestion Control. In *SIGCOMM* (1996).

[34] Mittal, R., Shpiner, A., Panda, A., Zahavi, E., Krishnamurthy, A., Ratnasamy, S., and Shenker, S. Revisiting Network Support for RDMA. In *SIGCOMM* (2018).

[35] Mittal, R., The Lam, V., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. TIMELY: RTT-Based Congestion Control for the Datacenter. In *SIGCOMM* (2015).

[36] Montazeri, B., Li, Y., Alizadeh, M., and Ousterhout, J. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *SIGCOMM* (2018).

[37] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the Social Network's (Datacenter) Network. In *SIGCOMM* (2015).

[38] Saeed, A., Dukkipati, N., Valancius, V., Contavalli, C., and Vahdat, A. Carousel: Scalable Traffic Shaping at End Hosts. In *SIGCOMM* (2017).

[39] Shreedhar, M., and Varghese, G. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking* (1996).

[40] Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S.-T., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., and McKeown, N. Programmable Packet Scheduling at Line Rate. In *SIGCOMM* (2016).

[41] Sultana, N., Galea, S., Greaves, D., Wójcik, M., Shipton, J., Clegg, R., Mai, L., Bressana, P., Soulé, R., Mortier, R., Costa, P., Pietzuch, P., Crowcroft, J., Moore, A., and Zilberman, N. Emu: Rapid Prototyping of Networking Services. In *ATC* (2017).

[42] Wilson, C., Ballani, H., Karagiannis, T., and Rowtron, A. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM* (2011).

[43] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM* (2015).

[44] Zhu, Y., Ghobadi, M., Misra, V., and Padhye, J. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *CoNEXT* (2016).

A Integrating Tonic within RDMA

Remote Direct Memory Access (RDMA) enables applications to directly access memory on remote endpoints without involving the CPU. To do so, the endpoints create a *queue pair*, analogous to a connection, and post requests, called *Work Queue Elements (WQEs)*, for sending or receiving data from each other's memory. While RDMA originated from InfiniBand networks, RDMA over Ethernet is getting more common in data centers [9, 22, 35]. In this section, we use RDMA to refer to RDMA implementations over Ethernet.

Once a queue pair is created, RDMA NICs can add the new "connection" to Tonic and use it to *on the sender side* to transfer data in response to different WQEs. Each WQE corresponds to a separate message transfer and therefore nicely fits Tonic's need for determining segment boundaries before starting data transmission.

For instance, in an RDMA Write, one endpoint posts a Request WQE to write to the memory on the other endpoint. Data length, data source address on the sender, and data sink addresses on the receiver are specified in the Request WQE. Thus, a shim layer between RDMA applications and Tonic can decide the segment boundaries and notify Tonic of the number of segments and the source memory address to read the data from on the sender. Once Tonic generates the next segment address, the rest of the RDMA NIC can DMA it from the sender's memory and add appropriate headers. An RDMA Send is similar to RDMA Write, except it requires a Receive WQE on the receiver to specify the sink address to which the data from the sender should be written. So, Tonic can still be used in the same way on the sender side.

As another example, in an RDMA Read, one endpoint requests data from the memory on the other endpoint. So, the responder endpoint should transmit data to the requester endpoint. Again, the data length, data source address on the responder, and data sink address on the requester are specified in the WQE. Thus, the shim layer can decide the segment boundaries and and transfer the data using Tonic.

Thus, Tonic can be integrated into RDMA NICs to replace the hard-coded transport logic on the sender-side of data transfer. In fact, two of our benchmark protocols, RoCE w/ DCQCN [43] and IRN [34], are proposed for RDMA NICs. That said, this is assuming there is a compatible receiver on the other side to generate the control signals (e.g., acknowledgments, congestion notifications, etc.) required by whichever transport protocol one chooses to implement on Tonic on the sender side.

While some implementations of RDMA over Ethernet such as iWarp [7] handle out-of-order (OOO) packets and implement TCP/IP-like acknowledgments, others, namely RoCE [8], assume a lossless network and have simpler transport protocols that do not require receivers to handle OOO packets and generate frequent control signals. However, as RDMA over Ethernet is getting more common in data centers, the capability to handle OOO packets on the receiver and generate various control signals for more efficient transport is being implemented in these NICs as well [34, 43].

Finally, Tonic provides in-order reliable data delivery within the same flow. Thus, messages sent over the same flow are delivered to the receiver in the same order. However, it is sometimes desirable to support out-of-order message delivery for a communication endpoint (e.g., a queue pair), for instance, to increase the performance of applications when messages are independent from each other, or when using "unconnected" endpoints (e.g., one sender and multiple receivers). It is still possible to support out-of-order message delivery using Tonic by creating multiple flows in Tonic for the same communication endpoint and using them concurrently. Extending Tonic to support out-of-order message delivery within the same flow is an interesting avenue for future work.

B High-Precision Per-Flow Rate Limiting

When using rate in the credit engine, if a flow with rate R bytes per cycle needs L more bytes of credit to transmit a segment, Tonic calculates $T = \lceil \frac{L}{R} \rceil$ as the time where the flow will have sufficient credit for transmission. It sets up a timer that expires in T cycles, and upon its expiry, queues up the flow in `ready-to-tx` for transmission (§3.3.2). Since Tonic cannot afford to do floating-point division within its timing constraints, R must be represented as an integer.

This creates a trade-off between the rate-limiting precision and the range of rates Tonic can support. If we represent R in bytes per cycle, we can compute the exact cycle when the flow will have enough credit, but cannot support rates lower than one byte per cycle or $\sim$1 Gbps. If we instead represent R in, say, bytes per thousand cycles, we can support lower rates (e.g., 1 Mbps), but $T = \lceil \frac{L}{R} \rceil$ will determine how many thousand cycles from now the flow can qualify for transmission. This results in lower rate conformance and precision for higher-bandwidth flows. As a concrete example, for a 20 Gbps flow, R would be 25000 bytes per thousand cycles. Suppose the flow has a 1500-byte segment to transmit. It will have enough credit to do so in 8 cycles but has to wait $\lceil \frac{1500}{25000} \rceil = 1$ thousand cycles to be queued for transmission.

Instead of committing to one representation for R, Tonic keeps multiple variables $R_1, \ldots, R_k$ for each flow, each representing flow's rate at a different level of precision. As the congestion control loop adjusts the rate according to network capacity, Tonic can efficiently switch between $R_1, \ldots, R_k$ to pick the most precise representation for computing T at any moment. This enables Tonic to support a wide range of rates without sacrificing the rate-limiting precision.

C Efficient Bitmap Operations

Tonic uses bitmaps as large as 128 bits to track the status of a window of segments for each flow. Bitmaps are implemented as ring buffers, with the head pointer corresponding to the first unacknowledged segment. As new acknowledgments arrive, the head pointer moves forward around the ring. To efficiently implement operations whose output depends on the values of *all* the bits in the bitmap, we must parallelize them by dividing the ring buffer into smaller parts, processing them in parallel, and joining the results. For large ring buffers, this divide and conquer pattern is repeated in multiple layers. As each layer depends on the previous one for its input, we must keep the computation in each layer minimal to stay within our 10 ns target.

One such operation finds the first set bit after the head. This operation is used to find the next lost segment for retransmission in the `marked-for-rtx` bitmap. The moving head of the ring buffer complicates the implementation of this operation. Suppose we have a 32-bit ring buffer A_{32}, with bits 5 and 30 set to one, and the head at index 6. Thus, $findfirst(A_{32}, 6) = 30$. We divide the ring into eight four-bit parts, "or" the bits in each one, and feed the results into an 8-bit ring buffer A_8, where $A_8[i] = OR(A_{32}[i : i+3])$. So, only $A_8[1]$ and $A_8[7]$ are set. However, because the set bit in $A_{32}[4 : 7]$ is *before* the head in the original ring buffer, we cannot simply use one as A_8's head index or we will mistakenly generate 5 instead of 30 as the final result. So, we need extra computation to find the correct new head. For a larger ring buffer with multiple layers of this divide and conquer pattern, we need to compute the head in each layer.

Instead, we use a lightweight pre-processing on the input ring buffer to avoid head index computation altogether. More specifically, using A_{32} as input, we compute A'_{32} which is equal to A_{32} except that all the bits from index zero to head (6 in our example) are set to zero. Starting from index zero, the first set bit in A'_{32} is always closer to the original head than the first set bit in A_{32}. So, $findfirst(A_{32}, 6)$ equals $findfirst(A'_{32}, 0)$ if A'_{32} has any set bits, and otherwise $findfirst(A_{32}, 0)$. This way, independent of the input head index H, we can always solve $findfirst(A, H)$ from two subproblems with the head index *fixed* at zero.

D Deciding C and T for Flows Using Tonic through the Socket API

In §5, we provide an example of how Tonic can be integrated into the Linux Kernel so that applications can use it through the Socket API. We introduce two parameters: (i) C, which is the flow's fixed segment size, and (ii) T, which is the duration that the Kernel waits for more data from the application before sending a "sub-segment" (a segment that is smaller than C) to Tonic. C and T can be configured for each flow based on its latency and throughput characteristics. For high-bandwidth flows, C can be set to MSS (or larger, if using TSO). For flows that only sporadically generate data segments, setting C and T, as we discuss below, is not as straightforward.

With a fixed C, increasing T results in more small segments being consolidated into C-sized segments before being sent to Tonic for transmission, but at the expense of higher latency. C determines the size of the segments and number of sub-segments generated by Tonic. Recall from §5 that a sub-segment is created when there is not enough data to make a full C-sized segment within T. Tonic needs all segments, except for the last sub-segment in a burst, to be of equal size. Thus, even if more data is added to the socket buffer after the sub-segment is sent to Tonic for transmission, Tonic has to successfully deliver all the previous segments before it can start transmitting the new ones. Thus, it is desirable to produce larger segments but fewer sub-segments. With a fixed T, increasing C results in larger segments. However, to produce fewer sub-segments, C should be picked such that in most cases, the data within a burst is divisible by C. Bursts are separated in time by T. So the choice of T affects the choice of C and vice versa.

For instance, if an application periodically generates 128-byte requests, C can be easily set to 128 and T based on the periodicity. As another example, for an application that periodically generates segments of widely-varying sizes, setting T to zero and C to the maximum expected segment size results in Tonic transmitting data segments as generated by the application without consolidation, potentially creating many sub-segments. For the same application, setting T to zero and C to the minimum expected segment size could result in Tonic generating many small segments as all segments will be broken into the minimum expected segment size. Note that these trade-offs become more pronounced if Tonic is to be used for flows that only sporadically generate data segments. For high-bandwidth flows, C can be set to MSS (or larger, if using TSO), and T depending on the application's traffic pattern and burstiness.

E Government Disclaimer

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA), NSF, or the U.S. Government.

F New Reno in Tonic

The following is the implementation of New Reno's loss detection and recovery algorithm on receipt of acknowledgments in Tonic [23]. Extra comments have been added for clarification.

```
module new_reno_incoming(
  /************************ INPUTS ****************************/
  // ACK, NACK, SACK, CNP, etc...
  input    [`PKT_TYPE_W-1:0]        pkt_type,
  input    [`PKT_DATA_W-1:0]        pkt_data_in,

  // Segment ID in the cumulative acknowledgment
  input    [`SEGMENT_ID_W-1:0]      cumulative_ack,

  // Segment ID that is selectively acknowledged, if any
  input    [`SEGMENT_ID_W-1:0]      selective_ack,

  // Number of segments acknowledged with the received acknowledgment
  input    [`WINDOW_INDEX_W-1:0]    newly_acked_cnt,

  // Segment ID at the beginning of the window, before and after the
  // acknowledgment
  input    [`WINDOW_INDEX_W-1:0]    old_wnd_start,
  input    [`WINDOW_INDEX_W-1:0]    new_wnd_start,

  // Current time in nanoseconds
  input    [`TIME_W-1:0]            now,

  //// Per-Flow State

  input    [`MAX_WINDOW_SIZE-1:0]   acked,
  input    [`MAX_TX_CNT_SIZE-1:0]   tx_cnt,
  input    [`SEGMENT_ID_W-1:0]      highest_sent,
  input    [`SEGMENT_ID_W-1:0]      wnd_start,
  input    [`WINDOW_SIZE_W-1:0]     wnd_size_in,
  input    [`TIEMR_W-1:0]           rtx_timer_amount_in,
  input    [`SEGMENT_ID_W-1:0]      total_tx_cnt,

  input    [`USER_VARS_W-1:0]       user_vars_in,

  /************************ OUTPUTS ***************************/
  output   [`FLAG_W-1:0]            mark_any_for_rtx,
  output   [`SEGMENT_ID_W-1:0]      mark_for_rtx_from,
  output   [`SEGMENT_ID_W-1:0]      mark_for_rtx_to,
  output   [`WINDOW_SIZE_W-1:0]     wnd_size_out,
  output   [`TIMER_W-1:0]           rtx_timer_amount_out,
  output   [`FLAG_W-1:0]            reset_rtx_timer,

  output   [`USER_VARS_W-1:0]       user_vars_out
);

/************************ Local Variables ********************
*
* Declarations ommited for brevity
*
**************************************************************/

/// is the ack new or duplicate?
assign is_dup_ack    = old_wnd_start == cumulative_ack;
assign is_new_ack    = new_wnd_start > old_wnd_start;

/// count duplicated acks
assign dup_acks      = is_new_ack ? 0:
                       is_dup_ack ? dup_acks_in + 1 : dup_acks_in;

// How many in_flight packets?
assign sent_out      = highest_sent - wnd_start;
assign in_flight     = sent_out - dup_acks;

// update previous highest ack
assign  prev_highest_ack_out = is_new_ack ? old_wnd_start : prev_highest_ack_in;

/// Should we do fast rtx?
assign do_fast_rtx = dup_acks == `DUP_ACKS_THRESH &
                      ((cumulative_ack > recover_in) |
                       (wnd_size_in > 1 & cumulative_ack - prev_highest_ack_in <= 4));

// if yes, update recovery sequence and updated ssh_thresh
assign recovery_seq_out = do_fast_rtx ? highest_sent : recovery_seq_in;
assign half_wnd       = in_flight > 2 ? in_flight >> 1 : 2;
assign ss_thresh_out  = do_fast_rtx ? half_wnd : ss_thresh_in;

//// if in recovery and this is a new ack, is it a
  // full ack or a partial ack? (Definition in RFC)
assign full_ack = is_new_ack & cumulative_ack > recover_in;
assign partial_ack = is_new_ack & cumulative_ack <= recover_in;

// mark for retransmission
assign mark_any_for_rtx = do_fast_rtx | partial_ack;

assign rtx_start = wnd_start_in;
assign rtx_end = wnd_start_in + 1;

// reset rtx timer if not in recovery
assign in_recovery_out = do_fast_rtx | (in_recovery_in & cumulative_ack <= recover_in);
assign reset_rtx_timer                = ~in_recovery_out;


assign in_timeout_out                 = (~full_ack) & in_timeout_in;

//// decide new window size

// keep a counter for additive increase
assign additive_inc_cntr_out          = in_recovery_out & ~in_timeout_in ? 0 :
                                        is_new_ack & wnd_size_in >= ss_thresh_in ?
                                              (additive_inc_cntr_in == wnd_size_in ? 0 :
                                               additive_inc_cntr_in + 1): additive_inc_cntr_in;


assign wnd_size_out = new_wnd_size >= `MAX_WINDOW_SIZE ? `MAX_WINDOW_SIZE : new_wnd_size;

always @(*) begin
  if (do_fast_rtx) begin
    // set it equals to new ss_thresh, expanded for performance reasons
    cwnd_out = sent_out - `DUP_ACKS_THRESH > 2 ? sent_out >> 1 : 1;
  end
  else if (~in_recovery_in & is_new_ack) begin
    if (cwnd_in < ss_thresh_out) begin
      cwnd_out = cwnd_in + newly_acked_cnt;
    end
    else if (wnd_inc_cntr_in >= cwnd_in) begin
      cwnd_out = cwnd_in + 1;
    end
    else begin
      cwnd_out = cwnd_in;
    end
  end
  else begin
    cwnd_out = cwnd_in;
  end
end
assign there_is_more = in_flight >= cwnd_in;

always @(*) begin
  if (do_fast_rtx) begin
    new_wnd_size = sent_out;
  end
  else if (~in_recovery_in & is_new_ack) begin
    new_wnd_size = cwnd_out;
  end
  else begin
    new_wnd_size = there_is_more ? sent_out : cwnd_in + dup_acks;
  end
end

//// break up user context into variables
assign {prev_highest_ack_in, in_recovery_in, recover_in,
        in_timeout_in, wnd_inc_cntr_in, ss_thresh_in,
        dup_acks_in, cwnd_in} = user_cntxt_in;

assign user_cntxt_out =  {prev_highest_ack_out, in_recovery_out, recover_out,
                          in_timeout_out, wnd_inc_cntr_out, ss_thresh_out,
                          dup_acks_outm, cwnd_out};


endmodule
```

FileMR: Rethinking RDMA Networking for Scalable Persistent Memory

Jian Yang[*] UC San Diego
Joseph Izraelevitz University of Colorado, Boulder
Steven Swanson UC San Diego

Abstract

The emergence of dense, byte-addressable *nonvolatile main memories* (NVMMs) allows application developers to combine storage and memory into a single layer. With NVMMs, servers can equip terabytes of memory that survive power outages, and all of this persistent capacity can be managed through a specialized NVMM file system. NVMMs appear to mesh perfectly with another popular technology, remote direct memory access (RDMA). RDMA gives a client direct access to memory on a remote machine and mediates this access through a memory region abstraction that handles the necessary translations and permissions.

NVMM and RDMA seem eminently compatible: by combining them, we should be able to build network-attached, byte-addressable, persistent storage. Unfortunately, however, the systems were not designed to work together. An NVMM-aware file system manages persistent memory as files, whereas RDMA uses a different abstraction — memory regions to organize remotely accessible memory. As a result, in practice, building RDMA-accessible NVMMs requires expensive translation layers resulting from this duplication of effort that spans permissions, naming, and address translation.

This work introduces two changes to the existing RDMA protocol: file memory region (FileMR) and range-based address translation. These optimizations create an abstraction that combines memory regions and files: a client can directly access a file backed by NVMM file system through RDMA, addressing its contents via file offsets. By eliminating redundant translations, it minimizes the amount of translations done at the NIC, reduces the load on the NIC's translation cache and increases the hit rate by 3.8× - 340× and resulting in application performance improvement by 1.8× - 2.0×.

1 Introduction

How scalable computer systems store and access data is changing rapidly, and these changes are in part motivated by the blurring of lines between traditionally separate system components. Nonvolatile main memory (NVMM) provides byte-addressable memory that survives power outages, blurring the line between memory and storage. Similarly, remote direct memory access (RDMA) allows a client to directly access memory on a remote server, blurring the line between local and remote memory. At first glance, by combining NVMM and RDMA, we could unify storage, memory and network to provide large, stable, byte-addressable network-attached memory. Unfortunately, the existing systems used to manage these technologies are simultaneously overlapping and incompatible.

NVMMs merge memory and storage. The technology allows applications to access persistent data using load/store instructions, avoiding the need for a block-based interface utilized by traditional storage systems. NVMMs are managed by an NVMM-aware file system, which mediates access to the storage media. With an NVMM-aware file system, applications can map a file into their address space, and then access it using loads and stores instructions, drastically reducing the latency for access to persistent data.

RDMA merges local and remote memory. RDMA allows a client to directly access memory on a remote server. Once the remote server decides to allow incoming access, it registers a portion of its address space as an RDMA *memory region* and sends the client a key to access it. Using the key, the client can enlist the server's RDMA network interface (RNIC) to directly read and write to the server's memory, bypassing the CPU. RDMA is popular as it offloads most of the networking stack onto hardware and provides close-to-hardware abstractions, exhibiting much better latency compared to TCP/IP protocol.

Ideally, we could combine NVMM and RDMA into a unified network-attached persistent memory. Unfortunately, NVM file systems and the RDMA network protocol were not designed to work together. As a result, there are many redundancies, particularly where the systems overlap in memory. Only RDMA provides network data transfer and only the NVMM file system provides persistent memory metadata, but both systems implement protection, address translation, naming, and allocation across different abstractions: for RDMA, memory regions, and for NVMM file systems, files. Naively using RDMA and NVMM file systems together results in a duplication of effort and inefficient translation layers between their abstractions. These translation layers are expensive, especially since RNICs can only store translations for limited amount of memory while NVM capacity can be extremely large.

In this paper, we present a new abstraction, called a *file*

[*]Now at Google

memory region (FileMR), that combines the best of both RDMA and NVM file systems to provide fast, network-attached, file-system managed, persistent memory. It accomplishes this goal by offloading most RDMA-required tasks related to memory management to the NVM file system through the new memory region type; the file system effectively becomes RDMA's control plane.

With the FileMR abstraction, a client establishes an RDMA connection backed by *files*, instead of memory address ranges (i.e., an RDMA memory region). RDMA reads and writes are directed to the file through the file system, and addressed by the file offset. The translation between file offset and physical memory address is routed through the NVMM file system, which stores all its files in persistent memory. Access to the file is mediated via traditional file system protections (e.g., access control lists). To further optimize address translation, we integrate a *range-based translation* system, which uses address ranges (instead of pages) for translation, into the RNIC, reducing the space needed for translation and resolving the abstraction mismatch between RDMA and NVMM file systems.

Our FileMR design with range-based translation provides a way to seamlessly combine RDMA and NVMM. Compared to simply layering traditional RDMA memory regions on top of NVMM, FileMR provides the following benefits:

- It minimizes the amount of translation done at the NIC, reducing the load on the NIC's translation cache and improving hit rate by 3.8× - 340×.
- It simplifies memory protection by using existing file access control lists instead of RDMA's ad-hoc memory keys.
- It simplifies connection management by using persistent files instead of ephemeral memory region IDs.
- It allows network-accessible memory to be moved or expanded without revoking permissions or closing a connection, giving the file system the ability to defragment and append to files.

The rest of this paper is organized as follows. Section 2 describes the necessary background on RDMA and NVMM file systems. Section 3 describes the design of the FileMR. Section 4 describes our proposed changes to RDMA stack and RNICs, and Section 5 introduce two case studies. Section 6 provides experimental results. Section 7 discusses the applicability of the FileMR on real hardware. Section 8 describes related work, and Section 9 concludes.

2 Background

This section introduces background on both RDMA and NVMM and describes the motivation for introducing a new memory abstraction for RDMA, detailing the issue of redundant memory management mechanisms and the reasons existing systems cannot solve this problem.

2.1 RDMA Networking

RDMA has become a popular networking protocol, especially for distributed applications [2, 20–22, 34, 36, 43, 47, 55, 56, 62]. RDMA exposes a machine's memory to direct access from the RDMA network interface (RNIC), allowing remote clients to directly access a machine's memory without involving the local CPU.

The RDMA hardware supports a set of operations (called *verbs*). *One-sided* verbs, for instance, "read" and "write", directly access remote memory without requiring anything of the remote CPU, in fact, these verb bypasses the remote CPU entirely. *Two-sided* verbs, in contrast, require both machines to post matching requests, for instance, "send" and "receive", which transfer data between registered buffers with addresses chosen by sender and receiver applications locally.

To establish an RDMA connection, an application registers one or more *memory regions* (*MRs*) that grant the local RNIC access to part of the local address space. The MR functions as both a name and a security domain: To give a client access to a region, the local RNIC supplies the MR's virtual address, size and a special 32-bit "rkey". Rkeys are sent with any one-sided verb and allow the receiving RNIC to verify the client has direct access to the region. For two-sided verbs, a send/recv operation requires both the sender and receiver to post matching requests, each attached to some local, pre-registered, memory region, negating the need for rkeys.

To manage outstanding requests, RDMA uses *work queues* derived from the virtual interface architecture (VIA) [10]. After establishing a connection, an application can initiate an RDMA verb through its local RNIC by posting work queue entries (WQEs). These entries are written onto a pair of queues (a queue pair or "QP"); one queue for send/write requests and one for read/receive requests. Once the entry is written to the queue pair, the RNIC will execute the RDMA verb and access the remote machine. Once the verb is completed, the RNIC will acknowledge the verb's success by placing a "completion" in the "completion queue" (CQ). The application can poll for the completion from the completion queue to receive notification that the verb completed successfully.

2.2 Nonvolatile Main Memory

Nonvolatile main memory (*NVMM*) is nonvolatile memory directly accessible via a load/store interface. NVMM is comprised of multiple nonvolatile DIMMs that are attached to the CPU memory bus and sit alongside traditional DRAM DIMMs. One or multiple nonvolatile DIMMs can be combined to form a single contiguous physical address space exposed to the OS [42].

As NVMM is a persistent media, it requires management software to provide naming, allocation and protection, and it is generally managed by a file system. However, unlike traditional file systems built for slower block devices, NVMM-aware file systems play a critical role in providing efficient NVMM access — the DRAM-comparable latency of NVMM means software overhead can dominate performance. As a result, NVMM-aware file systems [7,57,59,60] avoid software overhead along the critical path in two ways:

First, they support the direct access `mmap()` (*DAX-mmap*) capability. DAX-mmap allows applications to map NVMM files directly into their address spaces and to perform data accesses via simple loads and stores. This scheme allows applications to bypass the kernel and file system for most data accesses, drastically improving performance for file access.

However, NVMM resides within the memory hierarchy, which can cause complications since caches are not persistent but can hold data that the application wants to persist. To persist data, cached writes to NVMM must be followed by cache-line flush or clean instructions to ensure the data is actually written back to NVMM, and non-temporal writes can bypass the CPU caches entirely. A store fence can enforce the ordering of the writes and guarantee the data will survive a power failure.

2.3 Managing RDMA and NVMM

Userspace RDMA accesses and NVMM mmapped-DAX accesses share a critical functionality: they allow direct access to memory without involving the kernel. Broadly speaking, we can divide both NVMM file systems and RDMA into a data plane that accesses the memory and a control plane that manages the memory exposed to user applications. The data plane is effectively the same for both: it consists of direct loads and stores to memory. The control plane, in contrast, differs drastically between the systems.

For both RDMA and NVMM file systems, the control plane must provide four services for memory management. First, it must provide naming to ensure that the application can find the appropriate region of memory to directly access. Secondly, it must provide access control, to prevent an application from accessing data it should not. Thirdly, it must provide a mechanism to allocate and free resources to expand or shrink the memory available to the application. Finally, it must perform translation between application level names (i.e., virtual addresses, or memory and file offsets) to physical memory addresses. In practice, this final requirement means that both RDMA and NVMM file systems must work closely with the virtual memory subsystem.

Table 1 summarizes the control plane metadata operations for RDMA and NVMM. These memory management functionalities are attached to different abstractions in RDMA and NVMM file systems. For RDMA we use abstractions such as memory regions and memory windows, and for NVMM file systems we use files.

Role	RDMA / File System	FileMR
Naming	Both (Redundant)	FS Managed
Permissions	Both (Redundant)	FS Managed
Allocation	Both (Redundant)	FS Managed
Appending	Not Allowed	FS Managed
Remapping	Not Allowed	FS Managed
Defragmentation	Not Allowed	FS Managed
Translation	Both (Incompatible)	FS Managed
Persistence	FS Only	FS Managed
Networking	RDMA	RDMA
CPU-Bypass	RDMA	RDMA

Table 1: **Control plane roles for RDMA and NVMM.** This table shows the features provided by RDMA and NVMM vs. FileMR.

2.3.1 Naming

Names provide a hardware-independent way to refer to physical memory locations. In RDMA applications, the virtual address of a memory region, along with its "host" machine's location (e.g., IP address or GID) serves as a globally (i.e., across nodes) meaningful name for regions of physical memory. These names are transient, since they become invalid when the application that created them exits, and inflexible since they prevent an RDMA-exposed page from changing its virtual to physical address mapping while accessible. To share a name with a client that wishes to directly access it via reads and writes, the host gives it the metadata of the MR. For two-sided verbs (i.e., send/receive) naming is ad-hoc: the receiver must use an out-of-band channel to decide where to place the received data.

NVMM-based file systems use filenames to name regions of physical memory on a host. Since files outlive applications, the file system manages names independent of applications and provides more sophisticated management for named memory regions (i.e., hierarchical directories and text-based names). To access a file, clients and applications on the host request access from the file system.

2.3.2 Permissions

Permissions determine what processes have access to what memory. In RDMA, the RDMA contexts are isolated and permissions are enforced in two ways. To grant a client direct read/write access to a memory location, the host shares a memory region specific "rkey." The rkey is a 32-bit key that is attached to all one-sided verbs (such as read and write) and is verified by the RNIC to ensure the client has access to the addressed memory region. For every registered region, the RNIC driver maintains the rkey, along with other RDMA metadata that provides isolation and protection in hardware-accessible structures in DRAM.

Permissions are established when the RDMA connection between nodes is created, and are granted by the application code establishing the connection. They do not outlive the process or survive a system restart. For two-sided verbs protection is enforced by the receiving application in an ad-hoc manner: The receiver uses an out-of-band channel to decide what permissions the sender has.

Access control for NVMM uses the traditional file system design. Permissions are attached to each file and designated for both individual users and groups. Unlike RDMA memory regions and their rkeys, permissions are a property of the underlying data and survive both process and system restart.

2.3.3 Allocation

RDMA verbs and NVMM files both directly access memory, so allocation and expansion of available memory is an important metadata operation.

For NVMM file systems, the file system maintains a list of free physical pages that can be used to create or extend files. Creation of a file involves marshalling the appropriate resources and linking the new pages into the existing file hierarchy. Similarly, free pages can be linked to or detached from existing files to grow or shrink the file. Changing the size of DAX-mmap'd files is easy as well with calls to `fallocate` and `mremap`.

Creating a new RDMA memory region consists of allocating the required memory resources, pinning their pages, and generating the rkey. Note that although many RNICs are capable of handling physical addresses [32], the physical address of a memory region is often out of the programmer's control (it depends, instead, on the implementation of `malloc`), and the page is pinned once the region is registered, leading to a fragmented physical address space.

In addition, changing the mapping of a memory region is expensive. For example, to increase the memory region size, the host server needs to deregister the memory region, reregister a larger region, and send the changes to any interested clients. The `rereg_mr` verb combines deregistration and registration but still carries significant overhead. MPI applications with public memory pool often use memory windows to provide dynamic access control on top of a memory region. This approach does not blend with NVMM file systems since it still requires static mappings of the underlying memory region.

Alternatively, programmers can add another memory region to the connection or protection domain. However, as memory regions require non-negligible metadata and RDMA does not support multi-region accesses, this solution adds significant complexity.

This fixed size limitation also prohibits common file system operations and optimizations, such as appending to a file, remapping file content, and defragmentation.

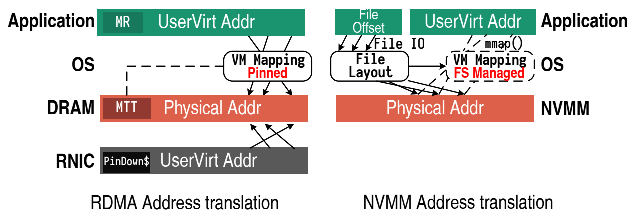


Figure 1: **Address translation for RDMA and NVMM.** RDMA (left) uses NIC-side address translation with pinning, while NVMM (right) allows the file system to maintain the layout of a file mapped to user address space.

2.3.4 Address Translation

RDMA and NVMM file system address translation mechanisms ensure that their direct accesses hit the correct physical page.

As shown in Figure 1, RDMA solves the problem of address translation by *pinning* the virtual to physical address, that is, as long as a memory region is registered, its virtual and physical addresses cannot change. Once this mapping is fixed, the RNIC is capable of handling memory regions registered on virtual address ranges directly: the RNIC translates from virtual addresses to physical addresses for incoming RDMA verbs. To do this translation, the NIC maintains a *memory translation table* (MTT) that holds parts of the system page tables.

The MTT flattens the translation entries for the relevant RDMA accessible pages and can be cached in the RNIC's onboard SRAM [54] to accelerate lookups of this mapping. The pin-down cache is critical for getting good performance out of RDMA — the pin-down cache is small (a few megabytes), a miss is expensive, and due to its addressing mechanism, most RNICs require all pages in a region be the same size. To circumvent these limitations, researchers have done significant work trying to make the most of the cache for addressing large memories [14, 22, 35, 36, 43, 48, 56, 62]. While complex solutions exist, the most common recommendation is to reduce the number of translations needed (e.g., addressing large contiguous memory regions with either huge pages or physical addresses).

The NVMM file system handles address translation in two ways, both different from RDMA. For regular reads and writes, the file system translates file names with offsets to physical addresses; this translation is done in the kernel during the system call. For memory mapped accesses, mmap establishes a virtual to physical address mapping from userspace directly to the file's contents in NVMM, loading the mapping into the page table. The file system only interferes on the page fault handling when a translation is missing between the user and physical addresses (i.e., a soft page fault); the file system is bypassed on normal data accesses.

The different translation schemes interfere with each other to create performance problems. If a page is accessible via

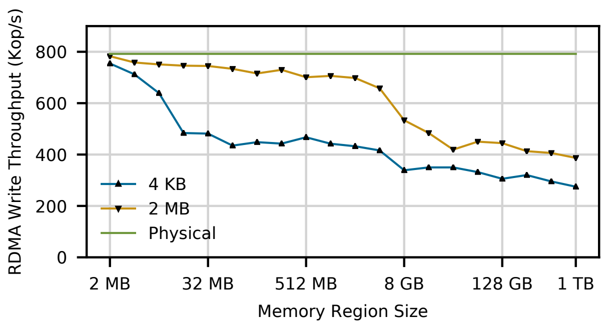


Figure 2: **RDMA Write performance over different memory region sizes.** This figure shows the throughput of 8-byte RDMA writes affected by the pin-down cache misses. Data measured on Intel Optane DC Persistent Memory with an Mellanox CX-3 RNIC.

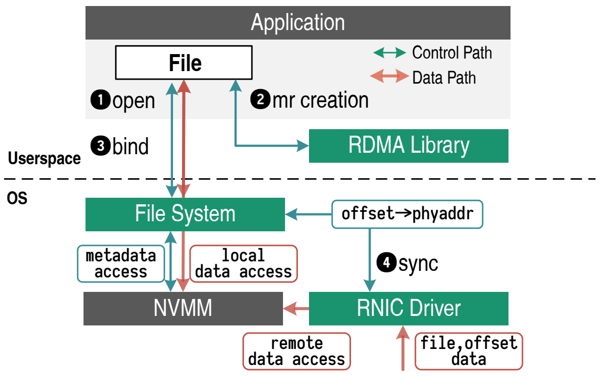


Figure 3: **FileMR: Control path and data path.** The user application communicates with the RDMA libraries and file system in control path, and access local and remote NVMM directly in datapath.

RDMA, it is pinned to a particular physical address, and furthermore, every page within the region must be the same size. Therefore the file system is unable to update the layout of the open file (e.g., to defragment or grow the file). As RDMA impedes defragmentation of files and prohibits mixing page sizes in RDMA accessible memory, memory regions backed by files must use many small pages to address large regions, overwhelming the pin-down cache and decimating RDMA performance.

Figure 2 shows the impact of pin-down cache misses on RDMA write throughput. Each work request writes 8 bytes to a random 8-byte aligned offset. When the memory region size is 16 MB, using 4 kB achieves 61.1% of the baseline (sending physical addresses, no TLB or pin-down cache misses) performance compared to 95.2% when using 2 MB hugepages. When the region size hits 16 GB, even 2 MB pages is not sufficient — achieving only 61.2% performance.

3 Design

FileMR is a new type of memory region that extends the existing RDMA protocol to provide file-based abstractions for NVMM. It requires minor changes to existing RDMA protocol and does not rely on any specific design of the file systems. FileMR can coexist with conventional RDMA memory regions, ensuring backward compatibility.

As shown in Table 1, the FileMR resolves the conflicts between RDMA and NVMM file systems that cause unnecessary restrictions and performance degradation through several innovations.

- **Merged control plane:** With an RDMA FileMR, a client uses a *file offset* to address memory, instead of a virtual or physical address. The FileMR also leverages the naming, addressing, and permissions of the file system to streamline RDMA access.
- **Range-based address translation:** The FileMR leverages the file system's efficient, extent-based layout description mechanism to reduce the amount of states the NIC must hold. As files are already organized in continuous extents, we extend this addressing mechanism to the RNIC, allowing the RNIC's pin-down cache to use a space efficient translation scheme to address large amounts of RDMA accessible memory.

The rest of this section continues as follows. We begin by describing the assumptions and definitions of FileMR, followed by the core mechanisms. Then, we describe the system architecture required to support our new abstraction.

3.1 Assumptions and Definitions

FileMR acts as an efficient and coordinated memory management layer across the userspace application, the system software, and the RDMA networking stack. This paper assumes the NVMM is actively managed by system software, and we describe it as a *file system*. Note that the concept of a file system is loosely defined: FileMR can be integrated with a kernel file system, a userspace file system, or a userspace NVMM library that accesses raw NVMM (also known as device-DAX) and provides naming, where a *file* maps to a corresponding entity.

This paper assumes NVMM is mapped to application address space in its entire lifecycle: As described in Section 2.2, the most prominent feature of NVMM is to have fine-grained persistency at a very low cost [63]. The design goal of FileMR is to enable remote NVMM accesses while retaining the simplicity and efficiency of local NVMM accesses. An alternative approach is to build holistic systems that manage both storage (NVMM) and networking (RDMA), these related work will be discussed in Section 8.2.

3.2 FileMR

Our new abstraction, the FileMR, is an RDMA memory region that is also an NVMM file. This allows the RDMA and NVMM control planes to interoperate smoothly. RDMA accesses to the FileMR are addressed by file offset, and the file system manages the underlying file's access permissions, naming, and allocation as it would any file. NVMM files are always backed by physical pages managed by the file system, so, when using a FileMR, the RDMA subsystem can simply reuse the translation, permission, and naming information already available in the file system metadata for the appropriate checks and addressing.

Figure 3 shows an overview of metadata and data access with FileMR. For metadata, before creating a FileMR, the application opens the backing file with the appropriate permissions (step ❶). Next, the application creates the FileMR (step ❷) and *binds* (step ❸) the region to a file, which completes the region's initialization. Binding the FileMR to the file produces a *filekey*, analogous to an rkey, that remote clients can use to access the FileMR. Once the FileMR is created and bound to a backing file, the file system will keep the file's addressing information in sync with the RNIC (step ❹).

For data access to a remote FileMR and its backing NVMM file, applications use the FileMR (with the filekey to prove its permissions) and a *file offset*. The RNIC translates between file offsets and physical addresses using translation information provided by the file system. In addition to one-sided read and write verbs to the FileMR, we introduce a new one-sided append verb that grows the region. When sending the append verb, the client does not include the remote address, and the server handles it like an one-sided write with address equal to current size of FileMR. It then updates the FileMR size and notifies the file system. As an optimization, to prevent faulting on every append message, the file system can preallocate translation entries beyond the size of a file. Even while the backing file is opened and accessible via a FileMR, local applications can continue to access it using normal file system calls or mmapped addresses — any change to the file metadata will be propagated to the RNIC.

3.3 Range-based Address Translation

NVMM file systems try to store file data in large, linear extents in NVMM. FileMR uses *range-based address translation* within the MTT and pin-down cache through a *RangeMTT* and *range pin-down* cache, respectively. This change is a significant departure from traditional RDMA page-based addressing. Unlike page-based translations, which translate a virtual to physical address using sets of fixed size pages, range-based translation (explored and used in CPU-side translation [5, 11, 23, 38]) maps a variably sized virtual address range to physical address. Range-based address translation is useful when addressing large linear memory regions (which NVMM file systems strive to create) and neatly leverages the

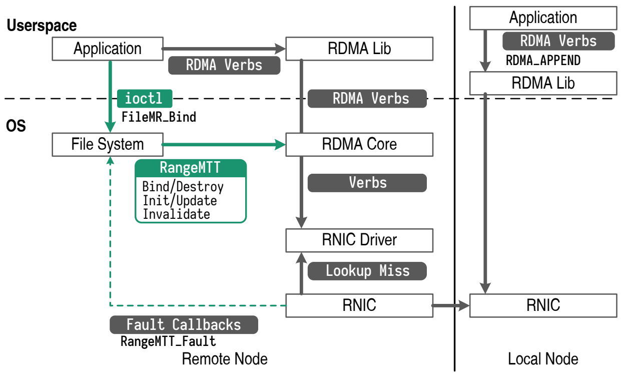


Figure 4: **Overview of FileMR components.** Implementing FileMR requires changes in file system, RDMA stack and hardware.

preexisting extent-based file organization.

For the FileMR, range-based address translation has two major benefits: both the space required to store the mapping and the time to register a mapping scale with the number of variable-sized extents rather than with the number of fixed size pages. Registering a page in the MTT and pin-down cache takes about 5 μs, this process requires locking memory descriptors and is hard to parallelize. As a result, a single core can only register memory at 770 MB/s with 4 kB pages. For NVMMs on the order of terabytes, the result registration time will be unacceptably long.

3.4 Design Overview

The implementation of the FileMR RDMA extension requires coordination and changes across several system components: the file system, the RNIC, the core RDMA stack, and the application. Figure 4 shows the vanilla RDMA stack (in grey) along with the necessary changes to adopt FileMR (in green).

To support the FileMR abstraction, the file system is required to implement the `bind()` function to associate a FileMR and a file, and, when necessary, notify the RDMA stack (and eventually the RNIC's RangeMTT and pin-down cache) when the bound file's metadata changes via callbacks (see Table 2). These callbacks allow the RNIC to maintain the correct range-based mappings to physical addresses for incoming RDMA requests.

Optionally, the file system can also register a set of callback functions triggered when RNIC cannot find a translation for an incoming address. This process is similar to on-demand paging [28, 29] and is required to support our new append verb, which both modifies the file layout and writes to it.

Supporting the FileMR abstraction also requires changes to the RNIC hardware. With our proposed RangeMTT, RNIC hardware and drivers would need to adopt range-based addressing within both the MTT and pin-down cache. Hardware range-based addressing schemes [5, 15, 23, 38] can be used to implement range-based address lookup. In our experiments

API	Description
`cm_bind()`	To notify RNIC of new bound file
`cm_init()`	To initialize RangeMTT entries
`cm_update()`	To update a RangeMTT entry
`cm_invalidate()`	To invalidate a RangeMTT entry
`cm_destroy()`	To destroy a file binding

Table 2: **File system to RNIC callbacks for FileMRs.** These callbacks are used by the file system to notify the RDMA stack and RNIC that file layouts (and consequently address mappings) have changed.

API	Description
`bind()`	Binds an opened file to FileMR
`ibv_reg_mr()`	Creates a FileMR with `FILEMR` flag
`ibv_post_send()`	Posts append w/ `APPEND` flag (`uverb`)
`ib_post_send()`	Posts append w/ `APPEND` flag (`kverb`)

Table 3: **New/changed RDMA methods.** These methods in the RDMA interface are new or have new flags under the FileMR system.

we simulate these changes using a software RNIC (see Section 4).

The FileMR also adds incremental, backwards compatible changes to the RDMA interface itself (see Table 3). It adds a new access flag for memory region creation to identify the creation of a FileMR. After its creation, the FileMR is marked as being in an unprovisioned state — the subsequent `bind()` call into the file system will allocate the FileMR's translation entries in the RangeMTT (via the `cm_bind` callback from the file system). The `bind()` method can be implemented with an `ioctl()` (for kernel-level file systems) or a library call (for user-level file systems). The FileMR also adds the new one-sided RDMA append verb. Converting existing applications to use FileMRs is easy as the applications only need to change its region creation code.

4 Implementation

We implemented the FileMR and RangeMTT for both the kernel space and userspace RDMA stack in Linux, and our implementations support the callbacks described in Table 2 and the changed methods in Table 3. The kernel implementation is based on Linux version 4.18, and userspace implementation is based on rdma-core (userspace) packages shipped with Ubuntu 18.04. Table 4 summarizes our implementation of FileMR with RangeMTT.

For our FileMR implementation on the NIC side, our implementation is based on a software-based RNIC: Software RDMA over Converged Ethernet (Soft-RoCE) [4, 25]. Soft-RoCE is a software RNIC built on top of ethernet's layer 2 and layer 3. It fully implements the ROCEv2 specification. Future research could work to build a FileMR compatible

	Item	Description
FileMR implementation on RDMA stack		
K	`ibcore`	Range-based TLB and FileMR kverbs
U	`libibverbs`	FileMR verbs in userspace
FileMR support on soft-RoCE		
K	`rxe`	Device driver and emu. RangeMTT.
U	`librxe`	Userspace driver
FileMR support for file system		
K	`nova`	a NVMM-aware file system
Applications adapting FileMR		
U	`novad`	Function stubs for remote file accesses
U	`libpmemlog`	NVMM log library

Table 4: **Summary of FileMR implementation.** This table shows the components modified to implement FileMR. The first column indicates the change is in kernel space (K) or userspace (U).

RNIC in real hardware.

To implement our RangeMTT, we followed the design introduced in Redundant Memory Mappings [23]: each FileMR points to a b-tree that stores offsets and lengths, and we use the offsets as indices. All RangeMTT entries are page-aligned addresses, since OS can only manage virtual memory in page granularity.

Unlike page-aligned RangeMTT, FileMR supports arbitrary sizes and allows sub-page files/objects. Each RangeMTT entry consists of a page address, a length field and necessary bits. These entries are non-overlapping and can have gaps for sparse files.

To support the append verb, the FileMR allows translation entries beyond its size. The append is one-sided but does not specify remote server addresses in the WR. On the server side, the RNIC always attempts to DMA to the current size of the FileMR and increases its size on success. When the translation is missing, the server can raise an IO page fault when IOMMU is available and a file system routine will be called to fulfill the faulty entries. Alternatively, if such support is unavailable, the server signals the client via a message similar to a receiver not ready (RNR) error.

Soft-RoCE manages the MTT entries as a flat array of 64-bit physical addresses with lookup complexity of O(1). We found similar design is implemented in hardware RNIC driver such as `mlx4`. For FileMR with a range pin-down cache miss, the entry lookup will traverse the registered data structures with higher time complexity (O(log(n))).

Soft-RoCE does not have a pin-down cache since the mappings are in DRAM. To emulate the RangeMTT, we built a 4096-entry 4-way associative cache to emulate the traditional pin-down cache, and a 4096-entry, 4-way associative range pin-down cache for FileMR. Each range translation entry consists of a 32 bit page address and a 32-bit length, which allows a maximal FileMR size of 16 TB (4 kB pages) or 8 PB (2 MB pages).

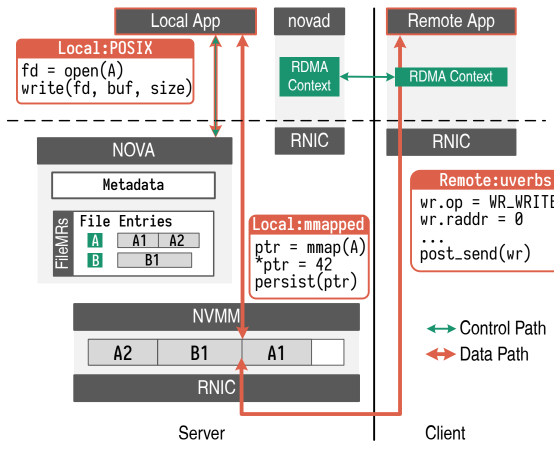


Figure 5: **Enabling remote NOVA accesses using FileMR.** Using FileMR, remote file accesses share a similar interface over RDMA as local NVMM accesses.

We adapted two applications to use FileMR. For a kernel file system, our implementation is based on NOVA [59], a full-fledged kernel space NVMM-aware file system with good performance. We also adapted the FileMR to libpmemlog, part of pmdk [40], a user-level library that manages local persistent objects, to build a remotely accessible persistent log.

5 Case Studies

In this section we demonstrate the utility of our design with our two case studies. In Section 5.1, we demonstrate how to use FileMR APIs to enable remote file accesses with consistent addressing for local and remote NVMM. In Section 5.2, we extend libpmemlog [40], a logging library designed for local persistent memory into a remotely accessible log, demonstrating how FileMR can be applied to userspace libraries.

5.1 Remote File Access in NOVA

In this section, we demonstrate an example usage of our FileMR by extending a local NVMM file system (NOVA [59]). By combining the NVMM file system, RDMA, and our new FileMR abstraction, we can support fast remote file accesses that entirely bypass the kernel.

NOVA is a log-structured POSIX-compliant local NVMM file system. In NOVA, each file is organized as a persistent log of variably sized extents, where the extents reside on persistent memory. The file data is allocated by the file system through per-cpu free lists and maintained as coalescing entries.

To handle metadata operations on the remote file system, we added an user-level daemon novad. The daemon opens the file to establish an FileMR, and receives any metadata updates (e.g. directory creation) from remote applications and applies them to the local file system.

On the client side, an application opens the file remotely by communicating with novad and receiving the filekeys. It can then send one-sided RDMA verbs to directly access remote NVMM. At the same time, applications running locally can still access the file with traditional POSIX IO interface, or map the file to its address space and issue loads and stores instructions.

Our combined system can also easily handle data replication. By using several FileMRs, we can simply duplicate a verb (with the same or different filekeys depending on the file system implementation) and send to multiple hosts, without considering the physical address of the files (so long as their names are equivalent).

5.2 Remote NVMM Log with libpmemlog

The FileMR abstraction only requires that the backing "file system" to appropriately implement the bind() method, RNIC callbacks, and have access to raw NVMM. For instance, a FileMR can be created by an application having access to the raw NVMM device. In this section, we leverage this flexibility and build a remote NVMM log based on libpmemlog.

We modify the allocator of libpmemlog to use the necessary FileMR callbacks — that is, whenever memory is allocated or freed for the log, the RNIC's RangeMTT is updated. The client appends to the log with the new append verb. On the server side, when the FileMR size is within the mapped RangeMTT, the RNIC can perform the translation while bypassing the server application. If not, a range fault occurs and the library expands the region by allocating and mapping additional memory.

6 Evaluation

In this section, we evaluate the performance of the FileMR. First, we measure control plane metrics such as registration cost, memory utilization of the FileMR, as well as the efficiency of RangeMTT. Then we evaluate application-level data plane performance from our two case studies and compare FileMR-based applications with existing systems.

6.1 Experimental Setup

We run our FileMR on servers configured to emulate persistent memory with DRAM. Each node has two Intel Xeon (Broadwell) CPUs with 10 cores and 256 GB of DRAM, with 64 GB configured as an emulated NVMM device. We setup Soft-RoCE on an Intel X710 10GbE NIC connected to a switch.

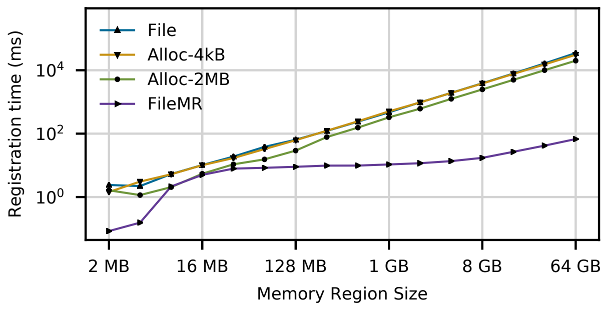


Figure 6: **FileMR registration time.** This figure shows the time consumed to register a fixed size memory region. The Y-axis is in log scale.

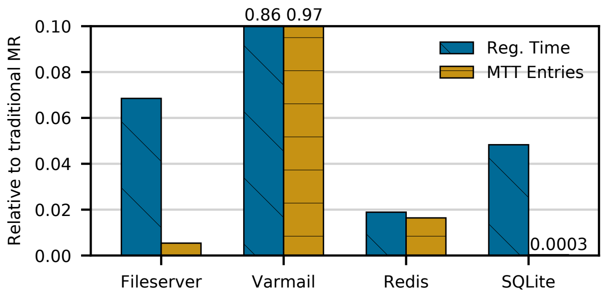


Figure 7: **FileMR on fragmented files.** Compared to traditional MRs, FileMR saves registration cost and RNIC translation entries.

Workload	# Files	Avg. Size	Description
Fileserver	7980	6.82 MB	File IOs
Varmail	4511	11.3 kB	Random IOs
Redis	2	561 MB	Write + Append
SQLite	1	109 MB	Write + Sync

Table 5: **Workload Characteristics.** Description of workloads to evaluate registration cost of FileMR and pin-down cache hit rate.

6.2 Registration Overhead

Allocated Regions We measure the time consumed in memory region registration using FileMRs versus conventional user-level memory regions backed by NOVA with 4 kB pages and anonymous buffers with 4 kB and 2 MB pages. This experiment demonstrates the use case when an application allocates and maps a file directly, without updating its metadata. For FileMR, we also include the time generating range entries from NOVA logs, which happens when an application opens the file for the first time.

As shown in Figure 6, registering a large size memory region consumes a non-trivial amount of time. It takes over 30 seconds to register a 64 GB persistent (**File**) and volatile (**Alloc-4K**) memory region with 4 kB pages. Using hugepages (**Alloc-2M**) reduces the registration cost to 20 seconds, while it only takes 67 ms for FileMR (three orders of magnitude lower). The FileMR registration time increases modestly as the region size grows mainly due to the internal fragmentation of the file system allocator.

For small files, NOVA only creates one or two extents for the file, while conventional MRs still interacts with the virtual memory routines of the OS, causing overhead.

Data Fragmentation FileMR benefits from the contiguity of the file data. The internal fragmentation of a file system can happen for two reasons: file system aging [19, 45] and using POSIX IO that changes file layout frequently. To evaluate the FileMR performance in a fragmented file system, we first warmed up the file system using four IO intensive workloads that issued POSIX IO requests: varmail and fileserver workloads in filebench [53], Redis [41] and SQLite [46] (using MobiBench [18]). Once the file system was warmed up and fragmented, we created memory regions over all files in NVMM. Table 5 summarizes the workloads.

As shown in Figure 7, running FileMR over the fragmented file still shows dramatic improvement on region registration time and memory consumption for MTT entries. Fileserver demonstrates the case with many files, where FileMR only creates 0.5% of the entries of traditional memory regions, and requires only 6.8% of the registration time. For a metadata-heavy workload (Varmail), FileMR only reduces the number of entries by 3% (due to the heavy internal fragmentation and small file size), but it still saves 20% on registration time because it holds the inode lock, which has less contention. Redis is a key-value store that persists an append-only file on the IO path, and flushes the database asynchronously — little internal fragmentation means that it requires 2% of the space and time of traditional memory regions. Similarly, SQLite also uses logging, resulting in little fragmentation, and drastic space and time savings.

6.3 Translation Cache Effectiveness

The performance degradation of RDMA over large NVMM is mainly caused by the pin-down cache misses (Figure 2). Since Soft-RoCE encapsulates RDMA messages in UDP and accesses all RDMA state in DRAM, we cannot measure the effectiveness of the cache through end-to-end performance.

Instead, we measure the cache hit ratio of our emulated pin-down cache and range pin-down cache for FileMR. We collect the trace of POSIX IO system calls for workloads described in Table 5, and replay them with one-sided RDMA verbs to a remote host.

Figure 8 shows the evaluation result. Our emulated range-based pin-down cache is significantly more efficient (3.8× - 340×) than the page-based pin-down cache. For large allocated files with a few entries, the range-based pin-down cache shows near 100% hit rate (not shown in figure).

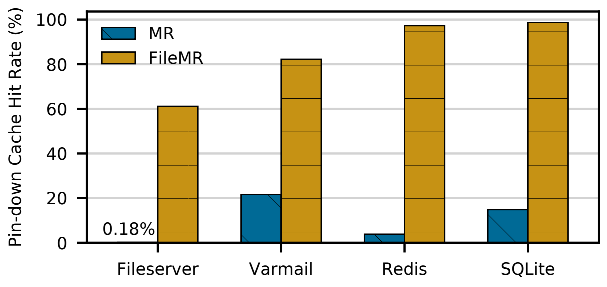


Figure 8: **Translation cache effectiveness.** FileMR significantly increases the effectiveness of the pin-down cache.

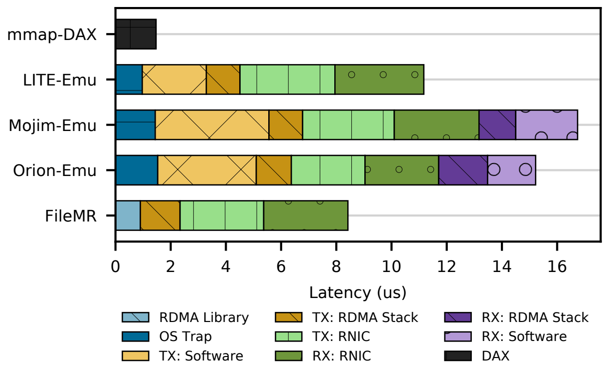


Figure 9: **Latency breakdown of accessing remote file.** FileMR can access remote file location without indirection on datapath.

6.4 Accessing Remote Files

To evaluate the datapath performance, we let a client access files on a remote server running novad (introduced in Section 5.1). The client issues random 1 kB writes using RDMA write verbs, and we measure the latency between the client application issuing the verb and the remote RNIC DMAs to the target memory address (`memcopy` for Soft-RoCE).

We compared FileMR with both mmapped local accesses and other distributed systems that provide distributed storage access. We implemented datapath-only versions of Mojim-Emu [64], LITE-Emu [56] and Orion-Emu [62] for Soft-RoCE. All these systems avoid translation overhead by sending physical addresses on the wire. We will further discuss these systems in Section 8.

In Figure 9, we show the latency breakdown of these systems. Note that the latencies of all systems are higher than a typical RDMA NIC, because Soft-RoCE is less efficient than a real RNIC. Also, we omit the latencies of UDP packet encapsulation and delivery, which dominate the end-to-end latency. It only takes 1.5 μs to store and persist 4 kB data to local NVMM. FileMR has lower latency than other systems because it eliminates the need for any indirection layer

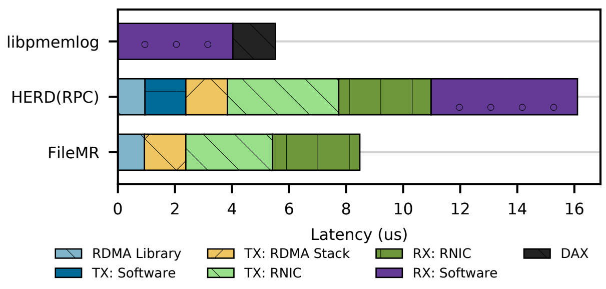


Figure 10: **Latency breakdown of accessing remote log.** With the append verb, Remote logging with FileMR achieves similar performance to local one.

(`msync()` system call for Mojim, shared memory write for LITE, and POSIX write for Orion).

6.5 Accessing Remote NVMM logs

Finally, we evaluate our introduction of the new append verb using our remote log implementation introduced in Section 5.2. We compare to a baseline `libpmemlog` on using local NVMM (bypassing the network), as well as with logging within the HERD RPC RDMA library [21, 22].

Figure 10 shows the latency breakdown of creating a 64 Byte log entry. It takes 5.5 μs to log locally with `libpmemlog`. FileMR adds 53% overhead for remote vs. local logging, while the HERD RPC-based solution adds 192% overhead.

7 Discussion

The current FileMR implementation relies on software-based RDMA protocols. In this section, we discuss the potential benefits and challenges of applying FileMR on hardware and other deeper changes to the RDMA protocol. We consider them to be the future work of this paper.

Data Persistence For local NVMM, a store instruction is persistent once data is evicted from CPU last-level cache (via cache flush instructions and memory fences). A mechanism called asynchronous DRAM refresh (ADR) ensures that the write queue on a memory controller is flushed to nonvolatile storage in the event of a power failure. There are no similar mechanisms in the RDMA world since ADR does not extend to PCIe devices. Making the task even more difficult, modern NICs are capable of placing data into CPU cache using direct cache access (DCA) [17], conceivably entirely bypassing NVMM.

The current workaround to ensure RDMA write persistency is to disable DCA and issue another RDMA read to the last byte of a pending write [9], forcing the write to complete and write to NVMM. Alternatively, the sender requests that the receiving CPU purposefully flush data it received; either embedding the flush request in an extra send verb or the immediate field of a write verb.

A draft standards working document has proposed adding a `commit` [50] verb to the RDMA protocol to solve the write persistency problem. A `commit` verb lists memory locations that need to be flushed to persistence. When the remote RNIC receives a commit verb, it ensures the all listed locations are persistent before acknowledging completion of the verb.

With the introduction of FileMR, implementing data persistence is simplified since there is no longer need to track modified locations at the client: the RNIC already maintains information about the files. A commit verb can simply request that all updates to a file are persisted, which is analogous to an `fsync` system call to a local file, which is usually light-weight. Even better, since the commit needs little state, a commit flag can be embedded to the latest write verb, reducing communication overhead.

Connection Management Several NVMM-based storage systems [30,43,56,62] store data across nodes, or use a model similar to distributed shared memory. This model requires establishing N^2 connections for N servers with NVMM. For user-level applications, the reliable connection transport enforces the protection domain within the scope of a process. Thus a cluster with N servers running p processes will establish $N^2 \times p^2$ connections.

Existing works [12, 12, 49] reduces this complexity by sharing queue pairs [49], multiplexing connections [27] or dynamically allocating connections [12] to reduce the RDMA states. These optimizations work well for MPI-based applications, but it is challenging to implement them for NVMM applications, especially for applications with fine-grained access control. In particular, a file system supports complex access control schemes, which may disallow sharing and multiplexing.

With the FileMR, the file permission is checked at the bind step, and so each server only requires a single connection to handle *all* file system requests, drastically reducing the amount of states required to store on the RNIC.

For NVMM, data replication is essential for reliability and availability. Existing RDMA-aware systems on distributed NVMM [6,33,43,62,64] transfer data multiple times to replicate NVMM because of the limitation (unreliable datagrams and two-sided verbs) of the existing RDMA protocol. The RDMA payload could be potentially multicasted by the current network infrastructure with FileMR, allowing a single RDMA verb to modify multiple copies of the same file.

Page Fault on NIC Some ethernet and RNICs support page fault or on-demand paging [28,29] (ODP). When using ODP, instead of pinning memory pages, the IOMMU marks the page as not present in IO virtual addresses. The RNIC will raise an interrupt to operating system when attempting DMA to a non-present page. The IO page fault handler then fills the entry with the mapping.

With ODP, a page fault is very expensive. In our experiment, it takes 475 μs to fulfill an IO page fault and complete a 8-byte RDMA write on a Mellanox CX-4 RNIC. In contrast, it only takes 1.4 μs to complete when the mapping is cached in the RNIC. In general, prefetching is a common way to mitigate the cost of frequent page faults. An optimization for ODP introduces `ioctl(advise_mr)` to hint prefetching [44], and recent research uses `madvise` system call to help prefetching local NVMM [8].

The design of FileMR is orthogonal to ODP, though it leverages the `append` verb. Fortunately, the file system is situated to provide better locality by prefetching ranges based on the file access pattern.

8 Related Work

The FileMR abstraction sits at the intersection of work in address translation, RDMA, and NVMM systems.

8.1 Address Translation

Reducing the cost of address translations has been the focus of work spanning decades. We here describe some common, general approaches and how they can be applied to RDMA and NVMM.

Using Hugepages Using hugepages is the standard way to reduce address transaction overhead and TLB pressure. In Linux, applications can explicitly allocate buffers from `libhugetlbfs`, which manages and allocates from a predefined page pool. An alternative is to allow the kernel to manage hugepages transparently using transparent hugepages (THP) [3, 37] or page swap [61] in the kernel, where the OS tries to allocate hugepages and merge smaller pages into hugepages (via compaction or swapping) in the background.

There are three drawbacks of using hugepages with RDMA and NVMM considered. First of all, applications [21,30,36] that use libraries such as `libhugetlbfs` will manage memory directly and bypass the file system. Second, transparent hugepage will violate the consistency of the MTT entries on the RNIC. Finally, since the current memory region uses a flat namespace, only one type of memory region is supported, which causes fragmentation when using hugepages. By introducing range-based translation, FileMR reduces the number of translation table entries significantly, while retain the support for file system managing the layout of the files.

Access Indirection Several existing works addressed the issue of accessing flat memory space by introducing an indirection layer for accesses and optimizing the communication cost.

LITE [56] uses physical memory region and lets all requests go into the kernel via shared memory. Remote Region [2] also redirects requests to kernel but consists of a pseudo-file system and a user library. Hotpot [43] and Mojim [64] use customized interfaces over memory mapped regions with support such as data replication and allocation. Storm [35] improves RDMA performance by introducing a new software stack that creates less RDMA states.

With the customized interface, these systems manage the data structures and RDMA states internally to reduce the state handled by RNICs. Programming with these interfaces is unintuitive, especially when working with an NVMM library that manages the data through memory-mapped files and handles allocations with its interface. Additionally, maintaining a physical memory region allows the remote server access arbitrary physical addresses, including DRAM.

Virtual Memory Contiguity As the size of the physical memory keeps increasing while the TLB size has grown slowly, several previous works discuss the contiguity of virtual memory address space. There are proposals on architecture support for coalesced [38], range [23] or segment [5] based address management, or accessing physical addresses with necessary permission checks [15]. In this paper, we assume the NIC hardware is capable of handling range-based entries using one of these mechanisms. Using software-based transparent page management [1, 61] can also increase the contiguity of virtual memory.

8.2 RDMA and NVMM

As discussed in the introduction, building systems that leverage the direct memory access of RDMA and NVMM is appealing. Significant work has already been completed.

Large RDMA Regions Creating memory regions over large memory with a flat namespace has become a popular choice for building RDMA-aware systems, even for those without a persistent memory component. Several systems use this strategy, including key-value stores [21, 34, 36], distributed memory allocators [2, 36, 55, 56], transactional systems, RPC protocols [20, 22, 47], and file systems [43, 62]. To better facilitate this use case, optimizations to the RDMA protocol such as on-demand paging [28, 29], dynamically connected transport [12], multi-path RDMA [31] have been purposed.

NIC Design PASTE [16] is a customized NIC designed for NVMM. It tightly couples the traditional networking stack with the NVMM file system. It provides a holistic design which performs naming and persistence in the networking stack. FileMR is designed for RDMA networking with NVMM file system, and supports general purpose verbs. FlexNIC [24] is a NIC that supports offloading software routines, such as key-value interface and packet classification down to the NIC. Floem [39] provides a programmable abstraction that describes the offload scheme. In contrary, FileMR focuses on a specific use case and can be further extended to support rich semantics.

Distributed File Systems Building distributed file systems by providing remote access at the file system level should provide remote access without interface changes. Orion [62] and Octopus [30] are two distributed NVMM-aware file systems. Orion is a kernel level file system that incorporates RDMA functionalities, and uses physical addresses for RDMA accesses. Octopus is a user level file system using FUSE [26] interface with hugepages to reduce page table entries. When using these file systems, all POSIX file accesses are intercepted and transferred with the file system routines.

There are two major issues in building distributed functionalities in the file system layer: the overhead of calling file systems routines and the granularity of access. In Linux, issuing system calls are expensive and involve multiple memory copies. In a DAX file system, the kernel still copies data from user buffers for security purposes. For mmapped data, the kernel supports "flushes" only in page granularity. These operations are expensive on large memories because the kernel needs to identify the dirty pages and persist them via memory flushes.

Large Memory and Persistent Connection This paper focuses on a particular way of utilizing RDMA networking: create memory regions over large, persistent memory with a flat namespace and maintain them for remote accesses during the application lifecycle. Alternative mechanisms for managing accessible RDMA, including using bounce buffers [52], registering a transient memory region for every access [51], and using tricks (DMA MR [62], Fast MR [58], FastReg MR [13]) require physical addresses in kernel space. We do not consider these solutions because they violate the file system's ability to manage the physical address space.

9 Conclusion

The conflicting systems on metadata management between NVMM and RDMA causes expensive translation overhead and prevents the file system from changing its layout. This work introduces two modifications to the existing RDMA protocol: the FileMR and range-based translation, thereby providing an abstraction that combines memory regions and files. It improves the performance of RDMA-accessible NVMMs by eliminating extraneous translations, while conferring other benefits to RDMA including more efficient access permissions and simpler connection management.

Acknowledgments

We thank our shepherd, Amar Phanishayee and the reviewers for their comments and suggestions. We would like to thank Tom Tapley for his insights and feedback, and Jiawei Gao for suggestions on improving the writing. This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 631–644. ACM, 2017.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, 2018.

[3] Andrea Arcangeli. Transparent hugepage support. In *KVM forum*, volume 9, 2010.

[4] InfiniBand Trade Association. SOFT-RoCE RDMA Transport in a Software Implementation, 2015.

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[6] Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. Rdmc: A reliable rdma multicast for large objects. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 71–82. IEEE, 2018.

[7] Dave Chinner. xfs: updates for 4.2-rc1. https://patchwork.kernel.org/patch/10723687/, 2015. Accessed 2020-1-1.

[8] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file I/O for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[9] Chet Douglas. RDMA with PMEM, Software mechanisms for enabling access to remote persistent memory. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/ChetDouglas_RDMA_with_PM.pdf. Accessed 2020-01-01.

[10] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The virtual interface architecture. *IEEE micro*, (2):66–76, 1998.

[11] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. Range translations for fast virtual memory. *IEEE Micro*, 36(3):118–126, May 2016.

[12] Richard Graham. Dynamically Connected Transport. https://www.openfabrics.org/images/eventpresos/workshops2014/DevWorkshop/presos/Monday/pdf/05_DC_Verbs.pdf. Accessed 2020-1-1.

[13] Sagi Grimberg. Introduce fast memory registration model (FRWR). https://patchwork.kernel.org/patch/2829652/. Accessed 2020-01-01.

[14] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215. ACM, 2016.

[15] Swapnil Haria, Mark D Hill, and Michael M Swift. Devirtualizing memory in heterogeneous systems. In *ACM SIGPLAN Notices*, volume 53, pages 637–650. ACM, 2018.

[16] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. Paste: A network programming interface for non-volatile main memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 17–33, 2018.

[17] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct cache access for high bandwidth network i/o. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 50–59. IEEE, 2005.

[18] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. AndroStep: Android Storage Performance Analysis Tool. In *Software Engineering (Workshops)*, volume 13, pages 327–340, 2013.

[19] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. Geriatrix: Aging what you see and what you don't see. a file system aging approach for modern storage systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 691–704, Boston, MA, July 2018. USENIX Association.

[20] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[21] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[22] Anuj Kalia Michael Kaminsky and David G Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference*, page 437, 2016.

[23] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 66–78, New York, NY, USA, 2015. ACM.

[24] Antoine Kaufmann, SImon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 67–81. ACM, 2016.

[25] Gurkirat Kaur and Manju Bala. Rdma over converged

ethernet: A review. *International Journal of Advances in Engineering & Technology*, 6(4):1890, 2013.

[26] http://fuse.sourceforge.net/.

[27] Matthew J Koop, Jaidev K Sridhar, and Dhabaleswar K Panda. Scalable mpi design over infiniband using extended reliable connection. In *2008 IEEE International Conference on Cluster Computing*, pages 203–212. IEEE, 2008.

[28] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page fault support for network controllers. *ACM SIGOPS Operating Systems Review*, 51(2):449–466, 2017.

[29] Liran Liss. On demand paging for user-level networking. https://openfabrics.org/images/eventpresos/workshops2013/2013_Workshop_Tues_0930_liss_odp.pdf. Accessed 2020-01-01.

[30] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, 2017. USENIX Association.

[31] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-path transport for rdma in datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 357–371, 2018.

[32] Mellanox. Physical Address Memory Region. https://community.mellanox.com/s/article/physical-address-memory-region/. Accessed 2020-01-01.

[33] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[34] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[35] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 97–108, New York, NY, USA, 2019. ACM.

[36] Tayo Oguntebi, Sungpack Hong, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. FARM: A prototyping environment for tightly-coupled, heterogeneous architectures. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 221–228, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 679–692. ACM, 2018.

[38] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–269. IEEE Computer Society, 2012.

[39] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: a programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, 2018.

[40] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[41] redislabs. Redis, 2017. https://redis.io.

[42] Arthur Sainio. NVDIMM: Changes are Here So What's Next. *In-Memory Computing Summit*, 2016.

[43] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337. ACM, 2017.

[44] Moni Shoua. Add support to advise_mr. http://oss.sgi.com/archives/xfs/2015-06/msg00478.html, 2018. Accessed 2020-1-1.

[45] Keith A Smith and Margo I Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 203–213, 1997.

[46] SQLite. SQLite, 2017. https://www.sqlite.org.

[47] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. DaRPC: Data center RPC. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[48] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster Than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15. ACM, 2017.

[49] Sayantan Sur, Lei Chai, Hyun-Wook Jin, and Dhabaleswar K Panda. Shared receive queue based scalable mpi design for infiniband clusters. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[50] Talpey and Pinkerton. RDMA Durable Write Commit. https://tools.ietf.org/html/draft-talpey-rdma-commit-00. Accessed 2020-01-01.

[51] T Talpey and G Kamer. High Performance File Serving With SMB3 and RDMA via SMB Direct. In *Storage Developers Conference*, 2012.

[52] Haodong Tang, Jian Zhang, and Fred Zhang. Accelerating Ceph with RDMA and NVMeoF. In *14th Annual OpenFabrics Alliance (OFA) Workshop*, 2018.

[53] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41, 2016.

[54] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 308–314. IEEE, 1998.

[55] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, and Thomas R Gross. Rstore: A direct-access DRAM-based data store. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 674–685. IEEE, 2015.

[56] Shin-Yeh Tsai and Yiying Zhang. LITE: Kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324. ACM, 2017.

[57] Matthew Wilcox. Add support for NV-DIMMs to ext4, 2014. https://lwn.net/Articles/613384/.

[58] Bob Woodruff, Sean Hefty, Roland Dreier, and Hal Rosenstock. Introduction to the InfiniBand core software. In *Linux symposium*, volume 2, pages 271–282, 2005.

[59] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[60] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[61] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: Operating system support for contiguity-aware tlbs. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 698–710, New York, NY, USA, 2019. ACM.

[62] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, 2019.

[63] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, FAST '20. USENIX Association, 2020.

[64] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.

TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10

Jaehyun Hwang Qizhe Cai Ao Tang Rachit Agarwal
Cornell University

Abstract

This paper presents design, implementation and evaluation of i10, a new remote storage stack implemented entirely in the kernel. i10 runs on commodity hardware, allows unmodified applications to operate directly on kernel's TCP/IP network stack, and yet, saturates a 100Gbps link for remote accesses using CPU utilization similar to state-of-the-art user-space and RDMA-based solutions.

1 Introduction

The landscape of cloud infrastructure has changed rapidly over the last few years. Two trends stand out:

- First, network and storage hardware has improved significantly, *e.g.*, network access link bandwidth has transitioned from 1Gbps to 40Gbps or even 100Gbps [4, 37]; and, fast non-volatile memory express (NVMe) storage devices that deliver more than a million input/output operations per second (IOPS) are being widely deployed [15, 23].
- Second, the need for fine-grained resource elasticity and high resource utilization has led to large-scale deployments of disaggregated storage [40, 43]; also see [9, 11, 12, 23, 24]. As a result, increasingly more applications now access storage devices over the network.

These changes have shifted performance bottlenecks back to the software stack — while network and storage hardware is able to sustain high throughput, traditional remote storage access stacks (that make remote storage devices available as local block devices, *e.g.*, iSCSI [35] and light-weight servers based on Linux) have unsustainable CPU overheads. For instance, traditional iSCSI protocol is known to achieve merely 70K IOPS per CPU core due to its high protocol processing and synchronization overheads [12, 15, 23, 24]. While this was not a bottleneck for slower storage devices and/or networks, saturating a single modern NVMe storage device now requires 14 cores, and saturating a 100Gbps link now requires 40 cores!

Responding to this challenge, both academic and industrial communities have been taking a fresh look at the problem of designing CPU-efficient remote storage stacks. Recently standardized NVMe-over-Fabrics (NVMe-oF), specifically, NVMe-over-RDMA [3, 30] keeps the kernel storage stack, but moves the network stack to the hardware. A complementary approach argues for moving the entire storage and network stack to the user space [24]. These proposals can achieve high performance in terms of IOPS per core, but require changes in applications and/or network infrastructure; such changes would be acceptable, if utmost necessary.

This paper explores a basic question: *"are infrastructure changes really necessary for efficient remote storage access"?* Exploring this question may help clarify our response to the challenges introduced by the two cloud infrastructure trends discussed above. An affirmative answer would make a strong argument for user-space stacks and/or specialized network hardware. However, if performance similar to above solutions can be achieved by re-architecting the kernel, then it changes the lens through which we view the design and adoption of software stacks in future: for example, rather than every organization asking *whether* to perform a ground-up redesign of their infrastructure, organizations that are already adopting user-space stacks and/or modern network hardware could ask *how* to port kernel remote storage stacks to integrate with their infrastructure. Thus, while the rest of the paper occasionally descends into kernel minutiae, the question we are asking has important practical implications.

Our exploration of the above question led to design and implementation of i10, a new *remote* storage stack within the kernel. i10 demonstrates that, at least for applications that are throughput bound, performance similar to state-of-the-art user-space and RDMA-based solutions can be achieved with minimal modifications in the kernel. i10 offers a number of benefits. First, i10 requires no modifications outside the kernel; thus, existing applications can operate directly on top of i10 without any modifications, whatsoever. Second, i10 operates directly on top of existing TCP/IP kernel protocol stack; thus, it requires no modifications in the network stack and/or hardware. Third, i10 complies with recently standardized NVMe-oF specification [3]; thus i10 can work with all emerging NVMe devices. Finally, over benchmark workloads, i10 saturates a 100Gbps link using a commodity server with CPU utilization similar to state-of-the-art user-space stacks and NVMe-over-RDMA products [24, 30]. The last benefit is perhaps the most interesting one as it fills a gaping hole in our understanding of kernel bottlenecks: as we discuss in Figure 1 and §4, existing bottlenecks for remote storage access are neither in the storage stack nor in the network stack; rather, the inefficiency lies at the boundary of the two stacks!

i10 achieves the above benefits using a surprisingly simple design that integrates two ideas (§2):

End-to-end dedicated resources and batching. Using dedicated resources and batching to optimize the software stack is a well-known technique, also used in CPU-efficient network stacks [6, 13, 19, 29]; however, as we will show, performance bottlenecks being at the boundary of storage and network stacks means that prior solutions of dedicating resources at per-core granularity and batching packets at socket level are no longer sufficient. i10 dedicates resources at the granularity of an i10-ℓane, creating a highly optimized pipe between each (core, target) pair, where target refers to the storage stack at the remote server (and not necessarily individual storage devices at the remote server). i10-ℓanes have the property that all control and data packets in any queue are destined to the same destination, and are to be transmitted on the same TCP session; thus, they can be efficiently batched at the entrance of the pipe to reduce network processing overheads (§2).

Delayed Doorbells. When accessing a local NVMe device, existing storage stacks "ring the doorbell" (signal the storage device about a new request) immediately upon receiving the request. i10 observes that while this is efficient for relatively low-overhead communication over PCI express, immediately ringing the doorbell leads to high context switching overheads for remote accesses where requests traverse over a high-overhead network stack and network fabric. i10, thus, introduces the idea of "delayed doorbells", where the worker thread in the storage stack delays ringing the doorbell until multiple requests are processed (or a timeout event happens). i10 also shows how the granularity of dedicated resources and batching in i10 interplays well with delayed ringing of doorbells to achieve the final performance (§2, §4).

i10 design simplicity also enables i10 implementation in Linux with modest modifications, while operating directly on an unmodified TCP/IP stack over commodity hardware[1]. i10 evaluation, over both benchmark workloads and real applications, demonstrates that i10 achieves throughput-per-core comparable to NVMe-over-RDMA [30]. Compared to state-of-the-art in-kernel remote storage stacks like NVMe-over-TCP [26], which was incorporated within the Linux kernel in March 2019, i10 both enables new operating points (*e.g.*, being able to saturate 100Gbps links) and also reduces the CPU utilization by 2.5× for already feasible operating points. Moreover, i10 maintains these benefits for all evaluated workloads including varying read/write ratios, request sizes and device types. Finally, i10 scales well with number of cores and with number of remote storage devices.

Going back to our starting point, i10 answers the original question about the necessity of application and/or network infrastructure changes for throughput-bound applications. Of course, this is already a large class of applications; however, batching used in i10 leads to the same latency-throughput tradeoff as in CPU-efficient network stacks [6, 13, 19, 29]: at low loads, latencies may be high (albeit, still at hundred-microsecond granularity and within 1.7× of NVMe-over-RDMA latency over storage devices). While not completely

[1]i10 implementation, along with documentation that enables reproducing all results, is available at https://github.com/i10-kernel/.

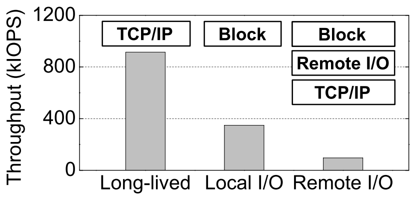


Figure 1: **Existing bottlenecks for remote storage access are at the boundary of the storage and network stacks.** The figure shows throughput-per-core for kernel network, local storage and remote storage stacks. We use 4KB random read requests between two servers with NVMe solid state drives connected via a 100Gbps link (detailed setup in §4). For long-lived flow, the network stack can sustain as much as ~30Gbps (roughly 915 kIOPS) per core using well-known optimizations (*e.g.*, TCP segmentation offload (TSO) and generic receive offload (GRO)); similarly, the storage stack for local I/O can sustain ~350 kIOPS per core. However, when integrated together with existing remote storage stacks, the achievable throughput reduces to 96 kIOPS (§4).

satisfying, our exploration has led us to believe that user-space stacks and RDMA-enabled solutions may be more useful for applications demanding absolutely minimal latency for each individual request. The question, however, remains open for such applications.

2 i10 Design

In this section, we describes the i10 design. We start with an overview (§2.1), followed by a detailed description of how i10 creates highly optimized i10-ℓanes using dedicated resources (§2.2), batching (§2.3) and delayed doorbells (§2.4). The next section provides some of the interesting implementation details for i10.

2.1 i10 Design Overview

i10 is designed and implemented as a shim layer between the kernel storage stack (specifically, the block device layer) and the kernel network stack. Figure 2 shows the high-level design of i10, including the i10 layer and end-to-end path between the host application and the target NVMe storage device. i10 does not control how applications are scheduled on the cores; each application may run on one or more cores, and multiple applications may share the same core. Applications submit remote read/write requests to the kernel through the standard read/write APIs; i10 requires no modifications to these APIs.

i10 achieves its goals using the core abstraction of an i10-ℓane— a dedicated pipe that is used to exchange both control and data plane messages along a set of dedicated resources. i10 creates i10-ℓanes and dedicates resources to each i10-ℓane at the granularity of (core, target)-pair, where target refers to the block device at the remote server (and not necessarily individual storage devices). For example, consider that (potentially more than one application running at) a core c submits read/write requests to two target servers `t1` and `t2`, each having multiple storage devices `d11, d12, ...` and `d21, d22, ...`. Then, i10 creates two i10-ℓanes, one `c` $\rightsquigarrow$ `t1` for all

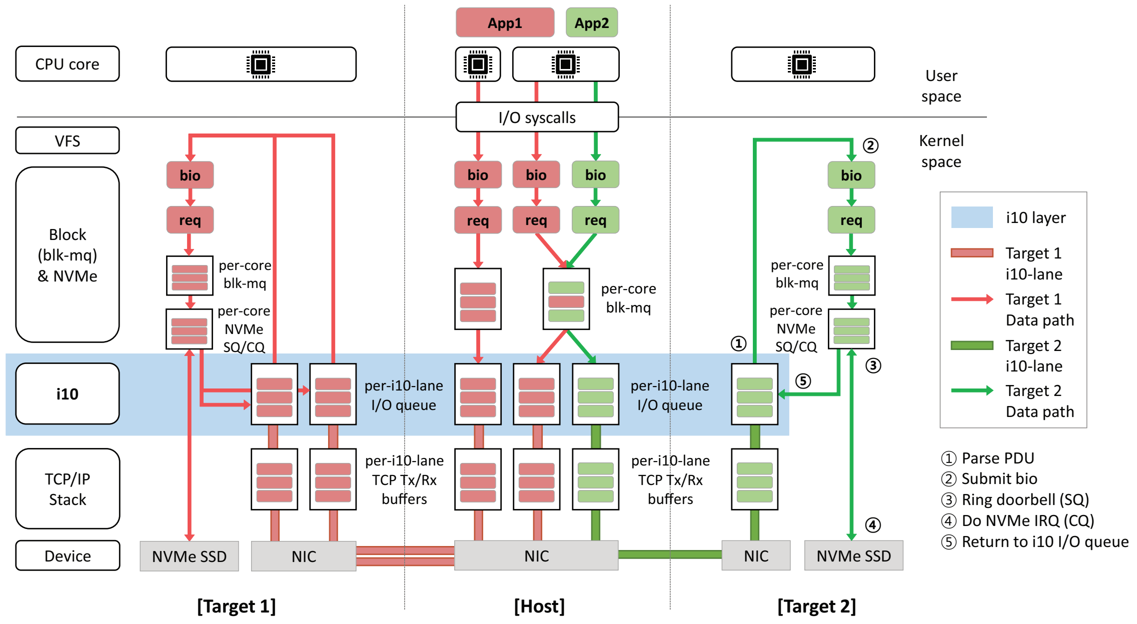


Figure 2: **End-to-end data path in i10 between one host and two target servers.** In this example, `app1` is sending read/write requests to `target1` using both core1 and core2, and `app2` is sending read/write requests to `target2` using core2. Thus, i10 creates three i10-ℓanes — one for each of (core1, target1), (core2, target1) and (core2, target2) pairs. Moreover, (green) requests submitted from `app2` are copied to the block layer request queue of core2 and (red) requests submitted from `app1` are copied to either the block layer request queue of core1 or core2 depending on which core the request comes from. While block layer request queues may contain requests going to different target servers (*e.g.*, the right block layer request queue at `host1`), each request is copied to the i10 I/O queue for i10-ℓane corresponding to the (core, target)-pair. Finally, if there were an `app3` running on `core2` sending requests to `target2`, it would completely share the i10-ℓane with `app2`.

requests going to the former set of storage devices and the other `c` $\rightsquigarrow$ `t2` for all requests going to the latter set of storage devices. Note that this is independent of the number of applications running on the core `c`. Moreover, if a single application running on two cores `c1 and c2` submits read/write requests to two target servers `t1` and `t2`, then i10 will create four i10-ℓanes, one for each (core, target) pair.

i10 uses three set of dedicated resources for each i10-ℓane (both at the host and at the target side). We first describe these dedicated resources and then discuss how they integrate into an end-to-end path between the host core and the target. The first dedicated resource is an I/O queue in the i10 layer (shown in the blue horizontal bar in Figure 2). The second is a dedicated TCP connection, along with its buffers, between the host and target i10 queues. Finally, a dedicated i10 worker thread for each core that interacts with i10 at the host side and for each core that is needed at the target side. Note that, for reasons that we will discuss in §2.2, i10 queues and TCP connections are at the per-lane granularity, and the i10 worker threads are at a per-core granularity.

We are now ready to describe the end-to-end path between the host core and the target. Upon receiving a request from a core, the block layer does its usual operations — generates a `bio` instance (that represents an in-flight block I/O operation in the kernel [28]), initializes the corresponding `request` instance using Linux kernel's support for multiple per-core block queues (`blk-mq`) [7, 14] and then, copies the request to the block layer's request queue for that core (these request queues are different from i10 queues). Finally, the block layer's `request` instance is converted to an i10 request; to be compliant with NVMe-oF standards, i10 requests are similar to a command Protocol Data Unit (PDU) [3]. Finally, using the context information within the block layer's request data structure, i10 requests are copied to the i10 queue for the corresponding i10-ℓane.

Having a dedicated queue for each i10-ℓane implies that all requests and data packets in a queue are destined to the same target server, and will be transmitted over the same TCP connection. Thus, i10 is able to batch multiple requests and data packets into i10 "caravans"[2], all to be processed and transmitted over the same TCP connection. This allows i10 to significantly reduce the network processing overheads by aggregating enough data to benefit from well-known optimizations like TSO and GRO. We discuss the precise details in §2.3.

i10 observes that the original NVMe specification was designed for accessing storage devices over PCI express (PCIe). Since PCIe provides a low-latency low-overhead communication between the storage stack and the local storage devices,

[2]i10 "caravans" are nothing but batches of requests; we use the term caravans to avoid confusion between i10 batches of requests and traditional batches — while traditional batches correspond to packets going to the same application port, i10 batches may be going to different storage devices, albeit within the same target server.

it was useful for the case of local access to "ring the doorbell" (provide a signal to the storage device that a new I/O request is ready to be served) immediately upon creating a request. However, in the case of remote accesses where requests traverse through a relatively high-latency high-overhead network, immediately ringing the doorbell leads to high context switching overheads for the worker threads. To alleviate these overheads, i10 introduces the idea of delayed doorbells, where the block layer worker thread processes multiple requests (or times out) before ringing the doorbell to wake up the i10 worker thread. This not only reduces the context switching overheads significantly, but also provides i10-ℓane queues with enough requests/data to generate right-sized caravans. We describe the precise mechanism in §2.4.

Finally, the i10 caravan is transmitted through the in-kernel socket interface. As shown in Figure 2, when the caravan arrives in the target-side i10 queue, i10 parses the caravan to regenerate the `bio` instances, corresponding requests and submits them to the block layer. Upon receiving the requests, the block layer executes the same steps as it does for accessing PCIe-connected local storage devices: the request is inserted to the NVMe submission queue and upon completion, the result is returned to the NVMe completion queue. After the local access, the result goes back to the block device layer, and finally is abstracted as a response caravan by i10 and sent back to the host server over the TCP connection.

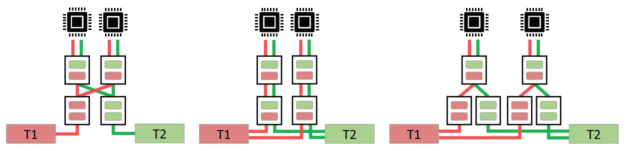
In the following three subsections, we present design details for the three building blocks of i10: i10-ℓane, i10 caravans, and delayed doorbells.

2.2 i10-ℓane

The two obvious options for creating i10-ℓane are (1) creating one i10-ℓane per target server, independent of the number of cores (Figure 3(a)); and (2) creating one i10-ℓane per core, independent of the number of target servers (Figure 3(b)). At high loads, the first option leads to high write contention among the block layer worker threads since they will need to write the `requests` to the same i10 queue. The second option is no better either — here, requests destined to different targets are forced to be in the same i10 queue resulting in preventing i10 caravans from batching enough requests, or high CPU overheads (for sorting requests to batch into the same caravans). Both these overheads become worse in the most interesting case of high-load regime. i10 avoids these overheads by creating an i10-ℓane for each (core, target)-pair (Figure 3(c)). That is, for applications that use P host cores to access data at T target servers, i10 creates $P \times T$ i10-ℓanes, independent of the number of storage devices at each target.

We now describe the resources dedicated to each i10-ℓane.

`blk-mq` level request queues. i10 exploits per-core request queue defined in the block layer (using `blk-mq` [7, 14]). Before the support for `blk-mq`, all block layer requests went to a single request queue per device. While queueing requests in a single queue could create head-of-line blocking, this design enabled scheduling so as to minimize the seek time on hard disks. For modern storage devices that support high-throughput random reads and writes using multiple cores, the equation is quite different due to two reasons: (1) multiple cores operating over a single queue becomes a performance bottleneck; and, (2) since seek time is not a problem, minimizing head-of-line blocking becomes more important. Thus, recent versions of Linux kernel enable the block layer to create per-core request queues and to maintain multi-queue context information for both `blk-mq` and underlying remote access layers. This enables i10 to efficiently demultiplex requests in `blk-mq` into individual i10-ℓane queues.



(a) per-target (b) per-core (c) per-target/per-core

Figure 3: **Creating i10-ℓane for each (core, target) pair is the right design.** The figure shows host cores, `blk-mq`, i10 queues, TCP connections, and target devices T1 and T2. For discussion, see §2.2.

i10 I/O queue. i10 creates one dedicated queue for each individual i10-ℓane. These queues are equivalent to the I/O queues from the NVMe standard [2] with the only difference that they communicate with a remote target server, not with local SSD devices. Once the requests from `blk-mq` are converted to NVMe-compatible command PDUs, they are inserted to i10 queues. The NVMe standard allows creating as many as 64K NVMe queues to enable parallel I/O processing; since we expect i10 to have no more than 64K simultaneously active i10-ℓanes at any server for most deployments, i10 design should not be limited by the number of available queues.

TCP socket instance. Each i10-ℓane maintains its own TCP socket instance and establishes a long lived TCP connection with the target, along with corresponding TCP buffers. The state needed to be maintained in the kernel for each individual TCP connection is already quite small. The only additional state that i10 requires is the mapping between the TCP connections and the corresponding i10-ℓane, which again turns out to be small (§5).

i10 worker thread. i10 creates a dedicated per-core worker thread whose responsibility is to conceptually move i10 caravans bidirectionally on i10-ℓanes. This worker thread starts executing when a doorbell is rung for any of the i10-ℓanes on the same core, and aggregates command PDUs in the queue of the corresponding i10-ℓane into caravans. The worker then moves the caravans to TCP buffers for the corresponding i10-ℓane. Finally, the worker thread goes into the sleep mode until a new doorbell is rung.

When the caravan reaches the target's TCP receive buffers,

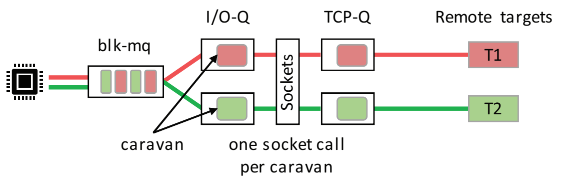


Figure 4: **Creating i10 caravans at the i10 queues is the right design for reducing per-request network processing overhead**. See discussion in §2.3.

the corresponding worker thread starts processing the requests in the caravan. First, it regenerates `bio` for each request in the caravan, followed by processing the requests as needed at the block layer. Upon receiving the signal from completion processing, the results are inserted into target's i10 queue and a caravan is created. It is not necessary for response caravans to have the same set of requests as in host caravans, because caravan's size can be different between host and target (§2.3).

2.3 i10 Caravans

Given that all requests in an i10 queue are going to the same destination over the same TCP connection, i10 batches multiple requests into an i10 caravan. This allows i10 to benefit from standard optimizations like TSO and GRO, which significantly reduces the network processing overheads. Our key insight here is that i10 queues are precisely the place to create caravans because of two reasons. First, at the block layer, the `blk-mq` is per-core and at any given point of time, may have requests belonging to different targets (as in Figure 3(c)); thus, batching the requests at the block layer would require significant CPU processing to sort the requests going to the same target device. Second, batching at the TCP layer would require i10 to process each request individually to send to TCP buffers, thereby creating one event per request; prior work [19] has shown that such per-request events results in high CPU processing overheads. By batching at its own queues, i10 reduces both of these overheads (Figure 4). We set the maximum amount of data carried by a caravan to be 64KB to align it with the maximum packet size supported by TSO. However, to prevent a single caravan from batching too many small-sized requests, each caravan may batch no more than a pre-defined aggregation size number of requests (§3).

2.4 Delayed Doorbells

The original NVMe specification was designed for accessing storage devices over PCI express (PCIe). Even though the standard itself does not prescribe how to use doorbells, the current storage stack simply rings the doorbell (that is, updates the submission queue doorbell register) whenever a request is inserted into the NVMe submission queue (Figure 5(a)). Since PCIe provides a low-latency low-overhead communication between the storage stack and the local storage devices, ringing the doorbell on a per-request basis reaches the maximum throughput of the device with minimal latency. However, in the case of remote accesses where requests traverse

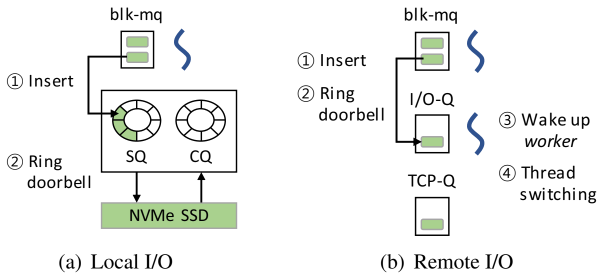


Figure 5: **Ringing the doorbell per request is effective for local PCIe-attached local storage, but not for remote storage access since the latter results in high context switching overhead.** See discussion in §2.4.

through a relatively high-latency high-overhead network, ringing the doorbell on a per-request basis results in high context switching overheads. In the specific context of i10, ringing the doorbell implies that as soon as the block layer thread inserts an i10 request to the i10 queue, it wakes up the i10 worker thread to handle the request immediately (Figure 5(b)). This incurs a context switch, which at high loads, could result in high CPU overheads (§4).

i10 alleviates these overheads using the idea of delayed doorbells. When an i10 queue is empty, a doorbell timer is set upon arrival of the first request. Then, the doorbell is rung either when the i10 queue has as many requests as a pre-defined aggregation size or when the timer reaches a timeout value, whichever happens earlier. Whenever the doorbell is rung, i10 caravans are created with all the requests in the i10 queue and the doorbell timer is unset. We note that delayed doorbells can be used independent of whether or not requests are batched into caravans. Moreover, this design will cause extra latency if applications generate low load (resulting in requests observing "timeout" amount of latency).

3 i10 Implementation Details

We implement i10 host and target in the Linux kernel 4.20.0. i10 implementation runs on commodity hardware (we do use the TSO and GRO features supported by most commodity NICs) and allows unmodified applications to operate directly on kernel's TCP/IP network stack. In this section, we discuss some interesting aspects of i10 implementation.

`kernel_sendpage()` vs. `kernel_sendmsg()`. There are two options to transmit i10 caravans via kernel socket interfaces. The first interface, `kernel_sendpage()` allows avoiding transmission-side data copy when sending each page of the data, but limits the aggregation size to be no more than 16. The second, `kernel_sendmsg()` takes a kernel I/O vector as a function argument and internally copies every scattered data of the I/O vector into a single socket buffer. This allows i10 aggregation size to be larger than 16, which leads to lower network processing overhead in some cases. Our tests reveal that `kernel_sendmsg()` achieves slightly better overall

CPU usage (that is, including data copy as well as network processing overheads), resulting in better overall throughput. Therefore, we use `kernel_sendmsg()` for i10 caravans to achieve better throughput.

i10 no-delay path. For latency-critical applications, it may be more important to avoid the latency incurred by i10 batching and delayed doorbells. For example, it may be desirable to execute a read request on file system metadata such as inode tables immediately upon submission so as to avoid blocking further read/write requests that cannot be executed without the response to the original request. For such cases, i10 supports a *no-delay path* — when such a latency-sensitive request arrives in i10 queue, i10 flushes all outstanding requests in the queue and processes the latency-sensitive request immediately. This is implemented using a simple check during the delayed doorbell ringing process: upon receiving a latency-sensitive request, the doorbell can be rung immediately, forcing i10 to create a caravan using all the outstanding requests along with the latency-sensitive request.

i10 parameters. In general, we expect throughput-per-core in i10 to improve with increasing aggregation size due to reduced network processing overheads. However, as we show in Appendix A in the technical report [16], increasing aggregation size beyond a certain threshold would result in marginal throughput improvements while requiring larger doorbell timeout values to be able to aggregate larger number of requests (thus, inflating per-request latency at low loads). This threshold — that is, the value that reaches the point of marginal improvements — of course, depends on kernel stack implementation. For our kernel implementation, we find 16 to be the best aggregation size with 50μs doorbell timeout value. We will use these parameters by default in our evaluation.

TCP buffer configuration. i10 caravans may be as large as 64KB. To this end, the TCP transmit buffer should have enough space for receiving caravans. However, TCP buffer size is generally adjusted by TCP's auto tuning mechanism — the Linux TCP implementation automatically increases the buffer size based on the bandwidth-delay product estimate for the transmit buffer, unless users specify a static buffer size via `setsockopt()`. Therefore, if the remaining transmit buffer is currently less than 64KB, the caravan would be fragmented even if kernel can provide more memory for the buffer resulting in higher processing overheads due to more than one socket call per caravan. For this reason, we explicitly use a fixed size of TCP buffers via `kernel_setsockopt()` at the session establishment stage. We set the buffer size to 8MB in this paper, which is sufficiently large to avoid caravan fragmentation.

4 i10 Evaluation

In this section, we evaluate an end-to-end implementation of i10. We first describe our evaluation setup (§4.1). We then start by evaluating i10 performance against state-of-the-art NVMe-over-RDMA (NVMe-RDMA) [30] and NVMe-over-TCP (NVMe-TCP) [26] implementations over a variety of settings including varying loads, varying number of cores, varying read/write request ratios and varying number of target servers (§4.2). Next, we use CPU profiling to perform a deep dive into understanding how different aspects of i10 design contribute to its performance gains (§4.3). Finally, we evaluate i10 over real applications (§4.4) and compare its performance with state-of-the-art user-space stacks (§4.5). Several additional evaluation results (including sensitivity analysis against aggregation size and doorbell timeout values, performance with variable request sizes, scalability with multiple applications sharing the same core, benefits of syscall batching, etc.) can be found in the technical report [16].

Table 1: **Experimental setup used in our evaluation.**

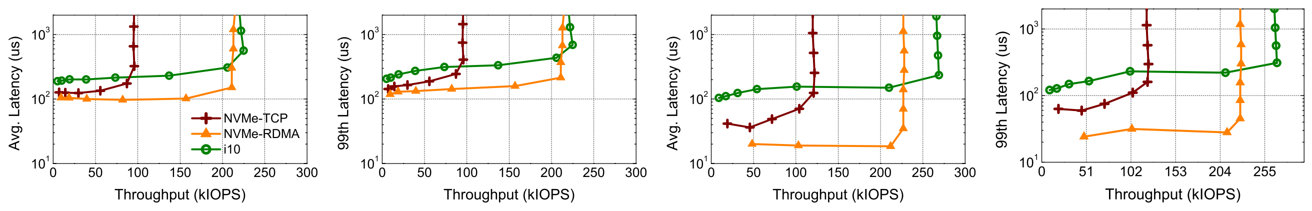
H/W configurations	
CPU	4-socket Intel Xeon Gold 6128 CPU @ 3.4GHz 6 cores per socket, NUMA enabled (4 nodes)
Memory	256GB of DRAM
NIC	Mellanox ConnectX-5 EX (100G) TSO/GRO=on, LRO=off, DIM disabled Jumbo frame enabled (9000B)
NVMe SSD	1.6TB of Samsung PM1725a
S/W configurations	
OS	Ubuntu 16.04 (kernel 4.20.0)
IRQ	`irqbalance` enabled
FIO	Block size=4KB, Direct I/O=on I/O engine=`libaio`, gtod_reduce=off CPU affinity enabled

4.1 Evaluation Setup

We use a testbed with two servers, each with 100Gbps links, directly connected without any intervening switches; while simple, this testbed allows us to ensure that bottlenecks are at the server-side thus allowing us to stress test i10. Both servers have the same hardware/software configurations (Table 1). Our NICs have one Ethernet port and one InfiniBand port, allowing us to evaluate both NVMe-TCP and NVMe-RDMA.

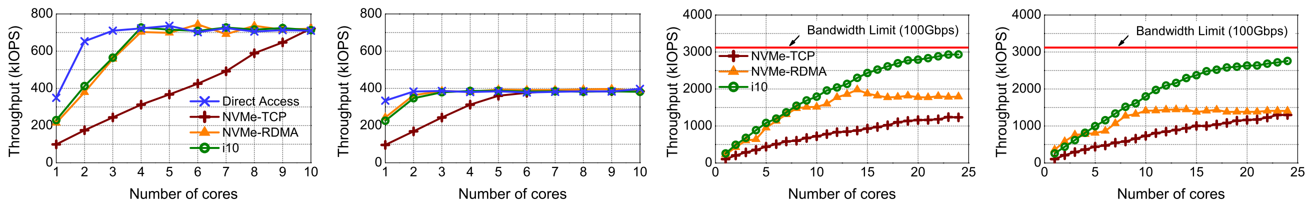
Our NICs support dynamically-tuned interrupt moderation feature that controls the network RX interrupt rate [41], which helps achieving maximum network throughput with minimum interrupts under heavy workloads; however, we disable it to show that i10 does not rely on special hardware features. Similarly, we do not optimize the Interrupt Request (IRQ) affinity configuration, but simply use the `irqbalance` provided by the OS. Finally, we use FIO [5] application for our microbenchmarks with a default I/O depth of 128. All I/O requests are submitted via the asynchronous I/O library (`libaio`) and direct I/O is enabled so as to bypass the kernel page cache and to make sure that I/O requests always go through the network to reach the target device.

i10 and NVMe-RDMA saturate our NVMe solid state



(a) Average Latency (SSD) (b) Tail Latency (SSD) (c) Average Latency (RAM) (d) Tail Latency (RAM)

Figure 6: **Single core performance (4KB random read)**: when compared to NVMe-TCP, i10 achieves significantly higher throughput-per-core with comparable latency; when compared to NVMe-RDMA, i10 achieves comparable throughput while achieving average and tail latency within 1.4× and 1.7×, respectively, for the case of SSDs.



(a) Read (SSD) (b) Write (SSD) (c) Read (RAM) (d) Write (RAM)

Figure 7: **Multi-core performance (4KB random read/write)**: i10 achieves throughput significantly better than NVMe-TCP and comparable to NVMe-RDMA while operating on commodity hardware; i10 and NVMe-RDMA also achieve near-perfect scalability with number of cores.

drives (SSD) with only 4 cores; hence, we also use RAM block device to evaluate multi-core scalability. In addition, our RAM-based experiments allow us to emulate i10 performance for the emerging Non-Volatile Main Memory devices as their performance is close to that of DRAM [45]. Unless otherwise stated, we use 4 cores for SSD-based and 16 cores for RAM-based evaluation.

4.2 Performance Evaluation

We now evaluate i10 performance across a variety of settings. All experiments in this subsection focus on host-side CPU utilization since target CPU was never the bottleneck (§4.3.2).

4.2.1 Single core performance

Figure 6 presents the single-core performance for all the evaluation schemes for both NVMe SSD and RAM block devices. The key takeaway here is that, when compared to NVMe-RDMA, i10 offers better throughput at high loads, but slightly higher latency (still at hundred-microsecond granularity) at low loads. Intuitively, when the load is high, i10 works with full-sized caravans without delayed doorbell timeouts, thus achieving high throughput. For low loads, i10 would wait for more requests to arrive in the i10 queue until the doorbell timer expires, which slightly increases its average end-to-end latency. However, we note that even for low loads, i10 average latency is comparable to that of NVMe-RDMA — *e.g.*, for the case of SSDs, where SSD access latency becomes the bottleneck, i10 achieves an average latency of 189μs while NVMe-RDMA achieves 105μs (Figure 6(a)). We observe similar trends for the tail (99th percentile) latency for the case of SSDs — as shown in Figure 6(b), i10's tail latency of 206μs is within 1.7× of NVMe-RDMA tail latency of 119μs, again since SSD access latency is the main bottleneck. For the case of RAM block device, i10 has higher average and tail latency compared to NVMe-RDMA since kernel overheads start to dominate; however, i10 still achieves comparable or better throughput per core. Finally, we also observe that all the three schemes perform better with RAM block device when compared to SSD since the former significantly reduces the access latency, not requiring any interrupt handling between device driver and RAM block device.

4.2.2 Scalability with number of cores

To understand i10 performance with increasing number of cores, we extend the previous single-core measurements to use up to 24 cores. Figure 7 presents the results. We observe that, for the case of random reads, i10 and NVMe-RDMA saturate the SSD with 4 cores, which is a factor 2.5× improvement over NVMe-TCP (Figure 7(a)), while direct access (where the requests go to local SSD) saturates the SSD using 3 cores. The case of random writes in Figure 7(b) is similar to Figure 7(a); both NVMe-RDMA and i10 saturate the maximum random write performance with 3 cores whereas NVMe-TCP requires 6 cores.

With RAM block device, i10 mostly outperforms both NVMe-TCP and NVMe-RDMA as shown in Figures 7(c) and Figures 7(d). Perhaps most interestingly, i10 is able to saturate the 100Gbps link, achieving 2.8M IOPS with ~20 cores. We observe that NVMe-RDMA stays around 1.5–2M IOPS after 10 cores for both random read and write workloads. While we believe that this performance saturation is not fundamental to NVMe-RDMA and is merely a hardware issue, we have not yet been able to localize the core problem; we note, however, that similar observations have been made

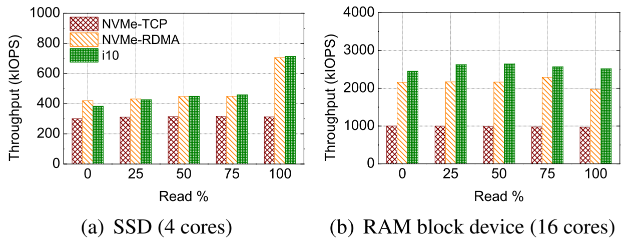


Figure 8: **i10 maintains its performance against NVMe-TCP and NVMe-RDMA with workloads comprising varying read/write ratios (4KB mixed random read/write)**.

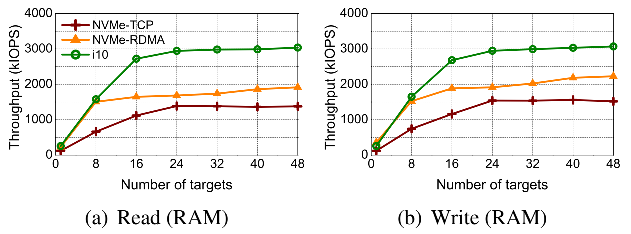


Figure 9: **i10 maintains its performance with increasing number of target devices (4KB random read and write).** The trend is similar to the multi-core/single-target case in Figure 7.

in other recent papers [31].

4.2.3 Performance with varying read/write ratios

Figure 8 presents results for workloads comprising varying ratio of read/write requests, varying from 0:100 to 100:0, for both SSD and RAM block device. For the case of SSD (Figure 8(a)), we observe that throughput in each case is limited by random write performance except that of the 100% read ratio case. This observation is consistent with the previous study that shows random write can interfere with random read because of wear leveling and garbage collection, where 75% read shows a similar IOPS with 50% read [24]. Consistent to results in previous subsections, both i10 and NVMe-RDMA saturate the SSD while requiring fewer cores than NVMe-TCP. With RAM block device (Figure 8(b)), the throughput changes with marginal fluctuation regardless of the read ratio; nevertheless, i10 continues to achieve comparable or better throughput than NVMe-RDMA across all workloads.

4.2.4 Scalability with multiple targets

We now evaluate i10 performance with increasing number of target devices, using up to 48 target RAM block devices. Here, we only focus on RAM devices since our testbed has a limited number of SSD devices. The setting here is that of each core running applications that access data from targets assigned to the application. The assignment is done in a round-robin manner — for up to 24 targets, each core will serve one target and for more than 24 targets, we assign the additional targets to the cores starting with core0 (*e.g.*, for 36 targets, the first 12 cores serve two targets each and the remaining 12 cores serve one target each).

In Figure 9, we observe that i10 outperforms both NVMe-TCP and NVMe-RDMA, saturating the 100Gbps link with 16 or more targets. The 24-target throughput is kept after 24 targets for all schemes as every host core is fully used (for i10, the 100Gbps link is already saturated). The overall trend is similar or even slightly better when compared to the RAM cases of the multi-core scalability scenario (Figure 7) as the I/O requests are processed in parallel across different i10-ℓanes. This result confirms that i10 maintains its performance benefits with increasing number of targets incurring little CPU contention across the various i10-ℓanes. For the scenarios where such CPU contention is severe (assuming an extremely large number of targets), the single-core/multi-target throughput is further studied in Appendix B in [16].

4.3 Understanding Performance Gains

We now evaluate how various design aspects of i10 contribute to its end-to-end performance, and then use CPU profiling to build a deeper understanding of i10 performance gains.

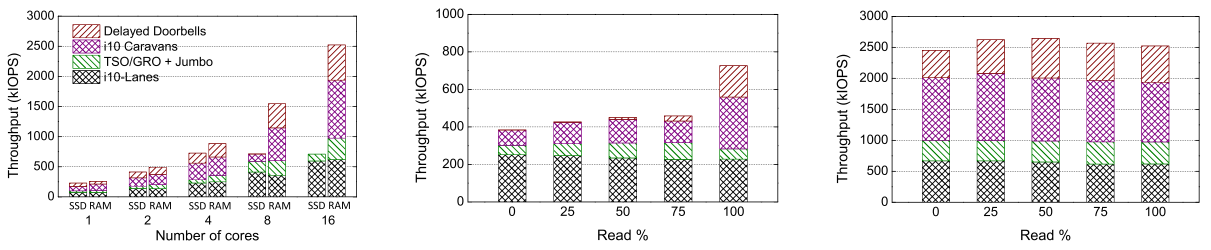
4.3.1 Performance contribution

Figure 10 shows that each of the design aspects of i10— i10-ℓane, i10 caravans, and delayed doorbells — are essential to achieve the end-to-end i10 performance. In Figure 10(a), we measure 4KB random read throughput increasing the number of cores from 1 to 16. The baseline is our i10-ℓane performance, which scales well with multiple cores. Enabling TSO/GRO and jumbo frames makes slightly further improvement over the i10-ℓane throughput. With NVMe SSD, i10 contribution is limited by the SSD performance (Figure 10(b)), but it is clear that with RAM block device, i10 caravans and delayed doorbells contribute to the performance improvement significantly, by 38.2% and 23.2% of the total throughput with 16 cores as shown in Figure 10(c). We also confirm that these improvement trends are maintained regardless of the read ratio for both SSD and RAM devices. The above results therefore indicate that all of the three building blocks are indeed necessary to design an end-to-end remote storage I/O stack that achieves the aforementioned benefits.

4.3.2 Understanding Bottlenecks

We now use CPU profiling for the 4-core case of SSD and the 16-core case of RAM block device in Figure 10(a), with the goal of understanding the bottlenecks for each of the three schemes. Our profiling results in Figure 12, Table 3 and Table 4 divide the entire remote access process into 7 components as described in Table 2. Our key findings are:

(1) i10 spends fewer CPU cycles in network processing: i10 improves upon NVMe-TCP by a factor of 2.7× in terms of CPU usage reduction in network processing parts (Network Tx and Rx combined) for both NVMe SSD and RAM block device, while showing comparable CPU utilization to NVMe-RDMA in the same parts. The main problem of NVMe-TCP is that it underutilizes the network capacity even with such high



(a) Throughput with varying # cores (b) Throughput (SSD) with varying read % (c) Throughput (RAM) with varying read %

Figure 10: **Each of i10-ℓane, caravans, delayed doorbells are necessary to achieve the end-to-end i10 performance**.

Table 2: **Taxonomy of CPU usage.**

Component	Description
Applications	Submit and receive requests/responses via I/O system calls (host). Ideally, all cycles would be spent in this component.
Block TX	Process the requests at the blk-mq layer and ring the doorbells to the remote storage access layer (host) or the local storage device (target).
Block RX	Receive the requests/responses from the network Rx queues or the local storage device.
Net TX	Send the requests/responses from the I/O queues (NVMe-RDMA uses Queue Pairs in the NIC).
Net RX	Process packets and insert into the network Rx queues (by the network interrupt handler).
Idle	Enter the CPU *Idle* mode.
Others	Include all the remaining overheads such as task scheduling, IRQ handling, spin locks, and so on.

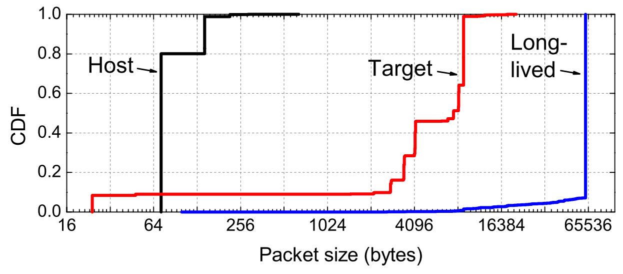


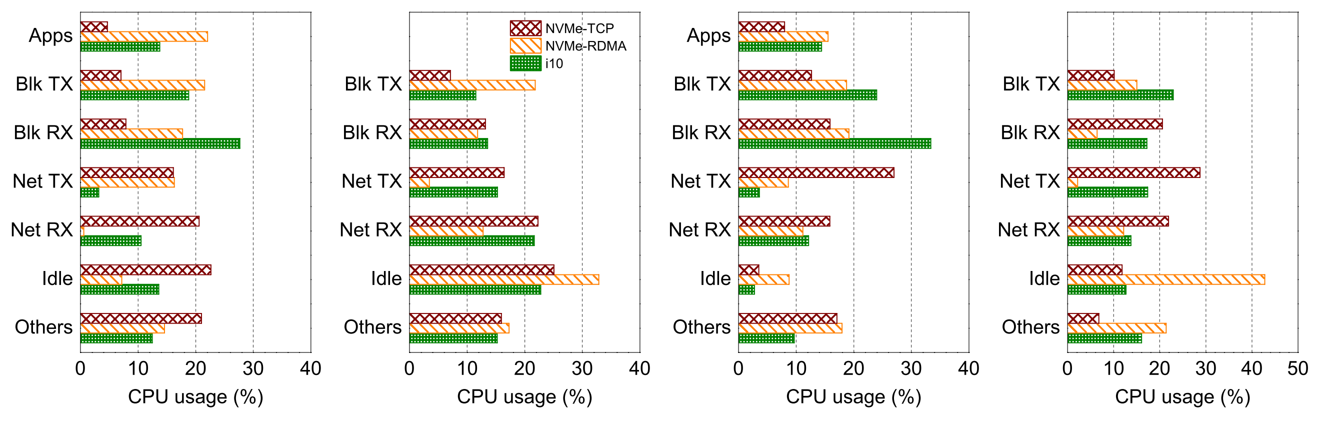
Figure 11: **For 4K random read, NVMe-TCP does not benefit from TSO** due to the packet sizes being mostly 72B at host and $<$ 9KB at target, much smaller than ideal 61KB packet size.

CPU usage in network processing. To investigate the reason further, we measure the packet sizes used in each TCP/IP processing for 4KB random read, comparing to a long-lived TCP connection that achieves ~30Gbps using a single core. Figure 11 reveals that about 80% of packet sizes generated by the host are 72 bytes, the I/O request PDU size of NVMe-TCP. This implies that almost every single small-sized request consumes CPU cycles for TCP/IP processing, increasing per-byte CPU usage. The target can generate a larger size of packets as it sends 4KB response data back to the host, but still most of them (98%) are under the jumbo frame size (9000 bytes) while the long-lived TCP connection generates mostly 61KB packets with TSO. i10 caravans help mitigate this bottleneck by generating 1152B packets (that is, 72B$\times$16) at the host and ~61KB packets at the target; i10 caravans are thus able to help reduce CPU usage by 30.12% and 31.14% for SSD and RAM block device in network processing parts, compared to the baseline i10 that uses only i10-ℓane.

(2) i10 minimizes context switching overheads: i10 achieves 1.7$\times$ CPU usage reductions over NVMe-TCP in *Others* part that includes task scheduling overheads. NVMe-TCP involves three kernel threads at the host — one for `blk-mq` that corresponds to the application thread, another for the remote I/O and TCP/IP Tx processing, and the third for the packet interrupt handling and TCP/IP Rx processing. This model avoids (i) slow responsiveness to other threads and/or (ii) long bottom half procedure for the incoming packet interrupts, but can incur high context switching overhead; our measurement indicates that each context switch takes 1–3μs per request, consuming more CPU cycles at the host. i10 amortizes this switching overhead using the idea of delayed doorbells; when compared to i10 design that does not use delayed doorbells, we observe reduction of CPU usage by 14.2% and 14.15% for NVMe SSD and RAM block device in the Others part. While the previous work mainly focuses on the target architecture [24], this host-side optimization turns out to be essential to improve the remote I/O throughput given that all the three remote storage access technologies consume more CPU cycles at the host regardless of the device type.

(3) i10 improves CPU efficiency allowing more cycles for applications: CPU resources saved in network processing and in context switching (using caravans and delayed doorbells) can be utilized by the applications, resulting in improved throughput per core. For instance, i10 allows applications to use 2.9$\times$ and 1.8$\times$ more CPU cycles for SSD and RAM devices, when compared to NVMe-TCP. NVMe-RDMA also shows 1.9$\times$ more CPU usage than NVMe-TCP on applications with RAM block device.

(4) i10 pushes the performance bottlenecks to the block layer: Tables 3 and 4 show that i10 pushes the performance bottlenecks from network processing and other lower layers (scheduling, etc., including in Others) to upper layers, making



(a) Host (SSD) (b) Target (SSD) (c) Host (RAM) (d) Target (RAM)

Figure 12: **CPU consumption at various system components for i10, NVMe-TCP and NVMe-RDMA**. i10 and NVMe-RDMA use significantly fewer CPU cycles for network processing and task scheduling (in Others) at the host while allowing applications to consume more CPU cycles, when compared to NVMe-TCP.

Table 3: **CPU usage contribution for 4KB random read with NVMe SSD**: i10 reduces CPU usages in network processing (Net TX and RX) using i10 caravans and in task scheduling (Others) using delayed doorbells, which allows more CPU cycles for applications and block layer. Here, i10-ℓane contribution is measured with enabling TSO/GRO and jumbo frames.

	Applications	Block TX	Block RX	Net TX	Net RX	Idle	Others
i10-ℓane	4.67	7.02	7.88	16.14	20.61	22.66	21.02
i10 Caravans	+9.85	+14.54	+16.41	-13.7	-16.42	-16.34	+5.66
Delayed Doorbells	-0.75	-2.76	+3.38	+0.73	+6.34	+7.26	-14.2
i10	13.77	18.8	27.67	3.17	10.53	13.58	12.48

Table 4: **CPU usage contribution for 4KB random read with RAM block device**. The trends are similar to the SSD case above.

	Applications	Block TX	Block RX	Net TX	Net RX	Idle	Others
i10-ℓane	8.0	12.68	15.9	27.0	15.84	3.51	17.07
i10 Caravans	+5.77	+10.88	+8.83	-24.79	-6.35	-1.07	+6.73
Delayed Doorbells	+0.63	+0.43	+8.66	+1.42	+2.67	+0.34	-14.15
i10	14.4	23.99	33.39	3.63	12.16	2.78	9.65

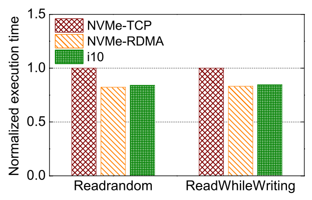
the block device layer a new bottleneck point in the kernel. i10 design did not attempt to perform changes in the block layer; however, it would be interesting to explore block layer optimizations to further improve end-to-end performance for remote (and even local) storage access.

We observe that RDMA still consumes a few CPU cycles to build and parse the command PDUs in network processing parts. In our profiling, one main difference between the SSD and RAM cases is that the RAM block device does not use IRQ to inform the I/O completion, but calls the the block layer functions immediately while in the SSD case, it still relies on the `nvme_irq` handling after the I/O is completed. This increases the IRQ handling overhead. Further, NVMe-RDMA generates another type of IRQ to call the block layer functions in the host after the network processing is done. This also slightly increases the IRQ handling overhead in the host.

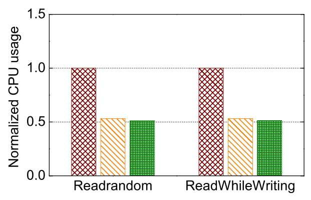
4.4 i10 performance with RocksDB

To evaluate i10 performance over real applications, we use RocksDB, a popular key-value store deployed in several production clusters [17]. We install RocksDB in the host server with a remote SSD device mounted with XFS file system. The RocksDB database and write-ahead-log files are stored on the remote device. To minimize the effect of the kernel page cache, we clear the page cache every 1 second during the experiment. We use `db_bench`, a benchmarking tool of RocksDB, for generating the two workloads using default parameters [18]: *ReadRandom* and *ReadWhileWriting*. Before running the workloads, we populate a 55GB database using the *bulkload* workload.

We measure the end-to-end execution time and kernel-side CPU utilization using a single core. In this experiment, the doorbell timeout value is set to 1ms as RocksDB is not an



(a) Execution time



(b) CPU usage in kernel

Figure 13: **i10 and NVMe-RDMA achieve** $1.2\times$ **lower latency and** $2\times$ **lower CPU utilization when compared to NVMe-TCP over SSD-based RocksDB.**

I/O bound application, and thus requires more time to aggregate appropriate number of requests. Figure 13 shows the performance of the three schemes, normalized by the NVMe-TCP performance. Again, we observe that i10 performs comparable to NVMe-RDMA while achieving almost $1.2\times$ improvements over NVMe-TCP in terms of execution time. The reason why this improvement is different with that of FIO benchmarks is that, RocksDB itself is the main CPU cycle consumer (up to 70% CPU usage) to perform data compression, key matching, etc. However, i10 still allows RocksDB to utilize more CPU resources by requiring $2\times$ lower CPU in the kernel across a fixed number of requests when compared to over NVMe-TCP. Our additional experiments with Filebench [39], presented in Appendix B in [16], indicate that if the application is I/O bound, i10 can achieve more than $2\times$ per-core throughput improvements over NVMe-TCP in a similar setup.

4.5 Comparison with ReFlex

Now we compare i10 with ReFlex, a user-level remote Flash access stack [24] using FIO. Unfortunately, despite significant effort, we were unable to install the remote block device kernel module for ReFlex [8] in our system[3], which is required for ReFlex host to run legacy Linux applications such as FIO. Thus, we make an indirect comparison with ReFlex, measuring i10 throughput using the same 10Gbps NICs used in [24] (that is, Intel 82599ES 10GbE NICs) and the same FIO script [8]. We also use 1-core i10 target as ReFlex target server uses 1-core per tenant. In this setup, i10 saturates the 10Gbps link with $\sim$3 cores, whereas ReFlex-FIO requires 6 cores according to [24]. This result still suggests that i10 would be a good option for remote storage I/O when we use legacy throughput-bound applications.

We also note that using IX [6]-based ReFlex clients achieves higher throughput; for instance, ReFlex reports achieving 850 kIOPS for 1KB read-only request using a single core [24], thus requiring only 2 cores to saturate a 10Gbps link. However, this requires the client server to run IX, precluding integration with unmodified legacy applications.

[3]The kernel module is based on an old version of kernel (4.4.0) that does not include relevant device drivers for our SAS SSD where the OS is installed. We also failed to boot with our NVMe SSD even with the up-to-date BIOS.

5 Discussion

We discuss a number of possible extensions for i10.

Can we apply i10 techniques to improve iSCSI performance? We believe that many of our optimizations may be useful to improve iSCSI [35] performance; for instance, the current Linux iSCSI implementation consumes CPU cycles inefficiently when sending I/O requests and responses, not fully utilizing TSO/GRO; moreover, it also runs a dedicated kernel thread for TCP/IP processing. i10 caravans and delayed doorbells alleviate precisely these bottlenecks, and thus may be useful for iSCSI.

Integrations with Emerging Transport Designs. i10 design and implementation was originally motivated by the question whether state-of-the-art performance for remote storage stacks can be achieved using simple modifications in the kernel. Thus, i10 design naturally integrates with existing network stack within the kernel. An intriguing future work would be to integrate i10 with emerging CPU-efficient network transport designs that use hardware offload.

Overheads of maintaining i10-ℓanes. Our i10-ℓane design does not introduce any additional overhead at either `blk-mq` or TCP/IP protocol layer, but simply exploits the existing per-core `blk-mq` that every request goes through regardless of whether it is remote access or not. The new overheads introduced by i10-ℓane are: i10 caravans, i10 queues, and TCP socket instances. To minimize memory usage and de-/allocation overheads of i10 caravan, each i10-ℓane maintains a single full-sized caravan instance (an array of kernel vectors that cover the aggregation size of requests and 64KB data) and reuse it whenever a new doorbell is rung. Memory requirement for maintaining per-(core, target) i10 I/O queues would be comparable to the current NVMe implementation, which creates per-(core, device) NVMe queues. Lastly, a dedicated socket requires small amount of state and thus, adds a minor memory overhead. We believe that such minor overheads are well worth the performance benefits achieved by i10.

Setting the right doorbell timeout value. In some extreme cases where a single host core needs to use many i10-ℓanes (that is, requests from a single core going to a large number of target servers), i10 may expose a throughput-latency tradeoff. Assuming the core generates requests at full rate, increasing number of i10-ℓanes means that the number of pending requests for each individual i10-ℓane will be reduced triggering doorbell timeout more often (see Appendix B in [16]). The tradeoff here is that to achieve throughput similar to the results in the previous section, we will now need larger delayed doorbell timeout value, resulting in relatively larger latency. This observation suggests that operators can set a higher timeout value when (1) applications are throughput bound; and/or (2) the number of targets per core increases. An interesting future work here is to explore setting the timeout value dynamically, depending on the "observed" load on the system.

Table 5: **Comparison of design decisions made in i10 with those in several prior works.**

	Storage stack	Network stack	API	App. event handling	Remote I/O event handling	Domain protection	Modification
Linux kernel	Kernel	Kernel	BSD socket	Syscalls	Per event	Native	No
MegaPipe [13]	Kernel	Kernel	lwsocket	Batched	per event	Native	App., Kernel
mTCP [19]	N/A	User-level	New API	Batched	N/A	Vulnerable	App., NIC driver
IX [6]	N/A	User-level	New API	Batched	N/A	Virtualization H/W	App., NIC driver
StackMap [44]	Kernel	Kernel	Ext. netmap	Batched	per event	Vulnerable	App., Kernel, NIC driver
ReFlex [24]	User-level	User-level	New API	Batched	Batched	Virtualization H/W	App., NIC driver
i10	Kernel	Kernel	BSD socket	Syscalls	Batched	Native	Kernel-only

6 Related Work

Table 5 compares i10 with several of the prior designs on optimizing the storage and network stacks. We briefly discuss some of the key related work below.

Existing remote storage I/O stacks. The fundamental performance bottlenecks of traditional remote storage stacks are well-understood [24]. For instance, iSCSI protocol [35] was designed to access remote HDDs over 1Gbps networks, achieving merely ∼70K IOPS per CPU core [12, 23, 24]. Similarly, a light-weight server for remote storage access based on Linux `libevent` and `libaio` achieves ∼75K IOPS per core [24]. Distributed file systems (*e.g.*, HDFS, GFS, etc.) are generally optimized for large data transfers, but are not so efficient for small-sized random read/write requests over high-throughput storage devices [10, 36]. i10 significantly improves throughput-per-core when compared to existing kernel-based remote storage stacks, achieving performance close to state-of-the-art user-space and RDMA-based products.

CPU-efficient network stacks within the kernel. Motivated by the fact that existing kernel *network* stacks were not designed for high network bandwidth links, there has been a significant amount of recent work on designing CPU-efficient network stacks [13, 27, 38, 42]. These stacks focus primarily on network stacks and are complementary to optimizing remote storage stacks; nevertheless, several optimization techniques (*e.g.*, syscalls batching/scheduling, per-core accept queue, etc.) introduced in these works may be useful for i10 as well. For instance, we demonstrate in Appendix B in [16] that syscall-level batching from [13, 38, 42], when integrated with i10, helps further improve the performance by 1.2×.

CPU-efficient user-space network stacks. The main motivation of this approach is that the kernel is extremely complex and high-overhead due to various overheads in system calls, process/thread scheduling, context switching, and so on. Prior studies reveal that the current kernel stack has limited network processing power in terms of the number of messages per second, so it is difficult to saturate network link capacity if the applications generate only small-sized messages, *e.g.*, under tens to hundreds of bytes [6, 19, 44].

To avoid these overheads, user-space solutions place the entire network protocol stack in the user space and directly access the NIC through the user-level packet I/O engines such as DPDK [1] and netmap [34], while bypassing the kernel [6, 19, 21, 22, 29, 32, 33]. By putting small-sized packets onto the NIC buffer directly in a batched manner, they significantly improve the messages per second performance. ReFlex [24] also implements the remote Flash access stack in the user space, on top of IX [6]. We note that modifying applications for using the user-level stacks might not be fundamental as recent new systems support the POSIX interfaces (*e.g.*, TAS [22], Strata [25], and SplitFS [20]). As we have discussed throughout the paper, i10's goals are not to beat the performance of user-space solutions but rather to explore whether similar performance can be achieved without modifications in the application and/or network infrastructure.

7 Conclusion

This paper presents design, implementation and evaluation of i10, a new in-kernel remote storage stack for high-performance network and storage hardware. i10 requires no modifications outside the kernel, and operates directly on top of kernel's TCP/IP network stack. We have demonstrated that i10 is still able to achieve throughput-per-core comparable to state-of-the-art user-space and RDMA-based solutions. i10 thus represents a new operating point for remote storage stacks, allowing state-of-the-art performance without any modifications in applications and/or network infrastructure.

Acknowledgements

We would like to thank our shepherd, Simon Peter, and the anonymous NSDI reviewers for their insightful feedback. We would also like to thank Amin Vahdat, Anurag Khandelwal, Saksham Agarwal and Midhul Vuppalapati for many helpful discussions during this work. This work was supported in part by NSF 1704742, ONR Grant N00014-17-1-2419, a Google Faculty Research Award, and gifts from Snowflake and Cornell-Princeton Network Programming Initiative.

References

[1] Intel DPDK: Data Plane Development Kit. https://www.dpdk.org/.

[2] NVM Express 1.4. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf, 2019.

[3] NVM Express over Fabrics 1.1. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf, 2019.

[4] Amazon. Introducing Amazon EC2 C5n Instances Featuring 100 Gbps of Network Bandwidth. https://aws.amazon.com/about-aws/whats-new/2018/11/introducing-amazon-ec2-c5n-instances/.

[5] Jens Axboe. Flexible IO Tester (FIO) ver 3.13. https://github.com/axboe/fio, 2019.

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI*, 2014.

[7] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *ACM SYSTOR*, 2013.

[8] Ana Klimovic et al. ReFlex github. https://github.com/stanford-mast/reflex, 2017.

[9] Peter Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *USENIX OSDI*, 2016.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *ACM SOSP*, 2003.

[11] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *USENIX NSDI*, 2017.

[12] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *ACM SYSTOR*, 2017.

[13] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *USENIX OSDI*, 2012.

[14] Christoph Hellwig. High Performance Storage with blk-mq and scsi-mq. https://events.static.linuxfound.org/sites/events/files/slides/scsi.pdf.

[15] HGST. LinkedIn Scales to 200 Million Users with PCIe Flash Storage from HGST. http://pcieflash.virident.com/rs/virident1/images/Case_Study_LinkedIn_PCIe_CS008_EN_US.pdf, 2014.

[16] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. Technical Report, https://github.com/i10-kernel/, 2020.

[17] Facebook Inc. RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/, 2015.

[18] Facebook Inc. RocksDB benchmark script. https://github.com/facebook/rocksdb/blob/master/tools/benchmark.sh, 2019.

[19] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX NSDI*, 2014.

[20] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *ACM SOSP*, 2019.

[21] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *ACM SoCC*, 2012.

[22] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *ACM Eurosys*, 2019.

[23] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *ACM Eurosys*, 2016.

[24] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *ACM ASPLOS*, 2017.

[25] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *ACM SOSP*, 2017.

[26] Lightbits Labs. The Linux Kernel NVMe/TCP support. http://git.infradead.org/nvme.git/shortlog/refs/heads/nvme-tcp, 2018.

[27] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *ACM ASPLOS*, 2016.

[28] Robert Love. *Linux Kernel Development*. Addison-Wesley, 3 edition, 2010.

[29] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network Stack Specialization for Performance. In *ACM SIGCOMM*, 2014.

[30] Kazan Networks. ACHIEVING 2.8M IOPS WITH 100GB NVME-OF. https://kazan-networks.com/blog/achieving-2-8m-iops-with-100gb-nvme-of, 2019.

[31] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: a fast transactional dataplane for remote data structures. In *ACM SYSTOR*, 2019.

[32] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*, 2019.

[33] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *ACM SOSP*, 2017.

[34] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[35] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). https://www.ietf.org/rfc/rfc3720.txt, 2004.

[36] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *IEEE MSST*, 2010.

[37] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM*, 2015.

[38] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *USENIX OSDI*, 2010.

[39] Vasily Tarasov. Filebench - A Model Based File System Workload Generator. https://github.com/filebench/filebench, 2018.

[40] Jason Taylor. Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond. In *OFC*, 2015.

[41] Mellanox Technologies. Dynamically-Tuned Interrupt Moderation (DIM). https://community.mellanox.com/s/article/dynamically-tuned-interrupt-moderation--dim-x, 2019.

[42] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. The Case for VOS: The Vector Operating System. In *USENIX HotOS*, 2011.

[43] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *USENIX NSDI*, 2020.

[44] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *USENIX ATC*, 2016.

[45] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *USENIX FAST*, 2019.

NetTLP: A Development Platform for PCIe devices in Software Interacting with Hardware

Yohei Kuga[1], Ryo Nakamura[1], Takeshi Matsuya[2], Yuji Sekiya[1]
[1]*The Univeristy of Tokyo,* [2]*Keio University*

Abstract

Observability on data communication is always essential for prototyping, developing, and optimizing communication systems. However, it is still challenging to observe transactions flowing inside PCI Express (PCIe) links despite them being a key component for emerging peripherals such as smart NICs, NVMe, and accelerators. To offer the practical observability on PCIe and for productively prototyping PCIe devices, we propose NetTLP, a development platform for software PCIe devices that can interact with hardware root complexes. On the NetTLP platform, software PCIe devices on top of IP network stacks can send and receive Transaction Layer Packets (TLPs) to and from hardware root complexes or other devices through Ethernet links, an Ethernet and PCIe bridge called a NetTLP adapter, and PCIe links. This paper describes the NetTLP platform and its implementation: the NetTLP adapter and LibTLP, which is a software implementation of the PCIe transaction layer. Moreover, this paper demonstrates the usefulness of NetTLP through three use cases: (1) observing TLPs sent from four commercial PCIe devices, (2) 400 LoC software Ethernet NIC implementation that performs an actual NIC for a hardware root complex, and (3) physical memory introspection.

1 Introduction

PCI Express (PCIe) is a widely used I/O interconnect for storage, graphic, network, and accelerator devices [16, 24, 25]. Not limited to connect the peripheral devices, some high-performance interconnects adopt the PCIe protocol [5, 22, 35]. Moreover, specifications of future interconnects are designed by extending the PCIe protocol [10, 14]. Such versatility of PCIe is derived from the packet-based data communication and the flexibility of PCIe topology. The PCIe specification defines building blocks comprising PCIe topologies: endpoint, switches, bridges, and root complexes. PCIe packets flow through point-to-point PCIe links between the blocks, and motherboard manufacturers can expand the PCIe topologies with these blocks depending on the use cases.

Table 1: Comparison of platforms for prototyping PCIe devices from the viewpoints of software and hardware.

		PCIe device	
		Software	Hardware
Root Complex	Software	QEMU	–
	Hardware	NetTLP	FPGA/ASIC

By contrast to the spread of PCIe, it is still difficult for researchers and software developers to observe PCIe and prototype PCIe devices, although they are crucial for optimizing performance and developing future PCIe devices. Observing PCIe transactions is difficult because PCIe transactions are confined in hardware. PCIe is not just a simple fat-pipe between hardware elements; it also has several features for achieving high-performance communication, i.e., hardware interrupt, virtualization, and CPU cache operations. Utilizing these features is important for exploiting PCIe efficiently; however, the concrete behaviors of the transactions in PCIe links cannot be determined unless special capture devices for observing PCIe hardware are used.

In addition to the observation, prototyping PCIe devices lacks productivity. Field Programmable Gate Array (FPGA) is a major platform for prototyping PCIe devices [26, 40, 44, 45, 50]. However, developing all parts of a PCIe device on an FPGA still requires significant effort, such as the great devotion of the NetFPGA project [52] for networking devices. Another approach is to adopt virtualization or simulation, e.g., GEM5 [11, 23] and RTL simulators [4, 31]. QEMU [9], which is a famous virtualization platform, can be used for prototyping PCIe devices from the software perspective. QEMU enables researchers and developers to prototype new hardware architecture; however, its environment is fully softwarized. QEMU devices can communicate with only the emulated root complex and cannot communicate with the physical root complex and other hardware connected to the root complex.

The goal of this paper is to bridge the gap between software and hardware for PCIe, as shown in Table 1. Our proposed platform, called NetTLP, offers softwarized PCIe endpoints that can interact with hardware root complexes. By using NetTLP, researchers and software developers can prototype their PCIe devices as software PCIe endpoints and test the software devices with actual hardware root complexes through the PCIe protocol. This hybrid platform of software and hardware simultaneously improves both the observability of PCIe transactions and the productivity of prototyping PCIe devices.

The key technique for connecting softwarized PCIe endpoints to hardware root complexes is to separate the PCIe transaction layer into software and put the software transaction layer on top of IP network stacks. Our FPGA-based add-in card, called NetTLP adapter, delivers Transaction Layer Packets (TLPs) to a remote host over Ethernet and IP networks. The substance of the NetTLP adapter is implemented in software on the IP network stack of the remote host with LibTLP, which is a software implementation of the PCIe transaction layer. The NetTLP platform consisting of the adapter and library enables software PCIe devices on IP network stacks to interact with hardware root complex through the NetTLP adapter. Moreover, TLPs delivered over Ethernet links can be easily observed by IP networking techniques such as tcpdump and Wireshark.

In this paper, we describe NetTLP, the novel platform for software PCIe devices. To achieve the platform, we investigate PCIe from the perspective of packet-based communication (§2) and then describe the approach to connect the software PCIe devices with hardware root complexes and process TLPs in software (§3) and describe its implementation (§4). In addition to micro-benchmarks (§5), we demonstrate three use cases of NetTLP (§6): observing behaviors of a root complex and commercial devices at a TLP-level, prototyping an Ethernet NIC in software interacting with a physical root complex, and physical memory introspection using NetTLP.

The contributions of this paper include the following:

- We propose a novel platform for prototyping PCIe devices in software, while the software devices can communicate with physical hardware such as root complexes, CPU, memory, and other PCIe devices. This platform offers high productivity for prototyping PCIe devices with actual interactions with hardware.

- We provide observability of PCIe transactions confined in hardware by the softwarized PCIe endpoints on the IP network stack. Our modified tcpdump can distinguish the encapsulated TLPs in Ethernet by NetTLP, enabling us to easily capture TLPs in an IP networking manner.

- We present detailed observation results of PCIe transactions with a root complex and commercial peripherals: an Intel root complex, Intel X520 10 Gbps NIC, Intel XL710 40 Gbps NIC, Intel P4500 NVMe, and Samsung PM1724 NVMe. The observation results by NetTLP reveal differences in their behaviors on PCIe transactions, for example, different usage of TLP *tag* fields.

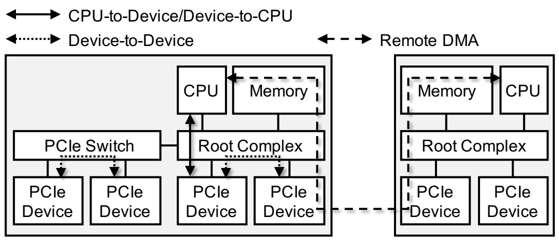


Figure 1: A PCIe topology and three communication models.

- We show a prototype of a nonexistent Ethernet NIC with NetTLP. This prototyping demonstrates the high productivity of the NetTLP platform; the NIC is certainly implemented in software, but it performs as an actual Ethernet NIC for a physical root complex.

- We demonstrate the possibility of developing memory introspection methods on NetTLP without implementing dedicated devices. As a proof-of-concept, we implemented two applications that gather process information from a remote host by DMA through NetTLP.

All source codes for hardware and software and captured data described in this paper are publicly available [1].

2 Background

PCIe is not only an interconnect, but also a packet-based data communication network. As with IP networks, PCIe has a layering model composed of a physical layer, a data link layer, and a transaction layer. The data link layer delivers PCIe packets across one hop over a PCIe link, while the transaction layer is responsible for delivering TLPs from a PCIe endpoint to a PCIe endpoint across the PCIe links. PCIe interconnect is composed of the following elements that are capable of supporting the layer functionalities: endpoints, switches, bridges, and root complexes. PCIe switches and root complexes route and forward PCIe packets in accordance with the addresses in memory-mapped I/O (MMIO) space or requester IDs. Any functionalities of PCIe stand at the packet-based communication, e.g., MSI-X for hardware interrupts is implemented by memory writes to specific memory addresses. Because of being such a packet-based network, PCIe topologies and their communication models are flexible, as depicted in Figure 1.

In IP networks, we can easily prototype and implement any part of the networks, such as end hosts, switches, and routers, and observe packets flowing in the networks; however, PCIe cannot do such things despite PCIe also being a packet-based network. PCIe was originally designed for I/O interconnects inside a computer; therefore, it is assumed that all the PCIe elements were implemented in hardware. This assumption and the current situation cause difficulty in investigating and developing PCIe and its elements.

For investigating PCIe, there are two major platforms: FPGA and QEMU. FPGA offers programmability on hardware for prototyping PCIe devices. By contrast, developing PCIe devices on FPGA still involves significant effort. Even when implementing a device, the device requires purpose-specific logic and various logic blocks such as PCIe core, DMA engine, memory controller, etc. Such functional blocks are not available, unlike software libraries. Moreover, we cannot observe the PCIe packets sent by the FPGA. We can see only part of the signals using logic analyzers, or expensive dedicated hardware. On the other hand, QEMU enables implementing PCIe devices on a full virtualized environment [15, 37]. However, QEMU does not implement the PCIe protocol, only DMA APIs. QEMU is used as a platform for researching new PCIe devices and discussing OS abstractions and implementations without real hardware and the PCIe protocol. Thus, QEMU PCIe devices cannot interact with the real host and hardware on the PCIe, although the features of root complexes have been evolving.

The two platforms have advantages and disadvantages: FPGA requires significant effort for prototyping and lacks observability, while QEMU devices cannot interact with hardware elements with the PCIe protocols. These disadvantages are because the platforms focus on only hardware or software. Root complexes and devices—the two major elements of PCIe—are hardware in FPGA or software in QEMU.

As a third platform, we advocate connecting software and hardware elements of PCIe. If PCIe devices are moved to software and connected to a hardware root complex, we achieve productive PCIe device prototyping in software and interactions with the real PCIe elements connected to the hardware root complex. Moreover, we can observe the PCIe transactions at the software PCIe device without resorting to dedicated hardware mechanisms. This relationship is similar to IP networks; IP network stacks at end hosts are software, while routers and switches are hardware.

3 NetTLP

To feasibly connect software PCIe devices and hardware root complexes, we propose to separate the transaction layer of PCIe into software, as illustrated in Figure 2. The transaction layer is responsible for the fundamental part of end-to-end PCIe communications: identifiers, i.e., memory addresses and requester IDs, routing, and issuing PCIe transactions. The softwarized transaction layer offers high productivity of PCIe device prototyping in software on top of the layer and observability of PCIe transactions by software.

To connect the softwarized transaction layer and the hardware data link layer, NetTLP has chosen a configuration that bridges a PCIe link and an Ethernet link. Because PCIe and Ethernet are packet-based networks, it is possible to deliver TLPs over Ethernet links by encapsulation. Once TLPs go to an Ethernet network, we can easily observe the TLPs like IP packets, implement the transaction layer in software, and prototype PCIe devices on top of software IP network stacks.

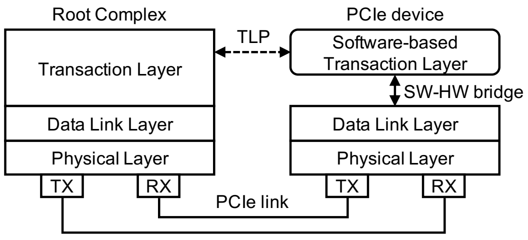


Figure 2: The layering model of PCIe and our approach that separates the transaction layer into software.

As with NetTLP, ExpEther [47, 48] and Thunderclap [29] also enable observing and manipulating TLPs. ExpEther extends PCIe links by delivering TLPs over Ethernet links. TLPs encapsulated by ExpEther would be observed on an Ethernet link between an ExpEther adapter and a hardware extension box in which peripheral devices are installed. In Thunderclap, Linux running on an ARM CPU on an FPGA processes TLPs with software. Similarly, some smart NICs can send and receive TLPs from CPUs on the NICs with abstracted DMA APIs [30, 33].

In contrast to the existing technologies, NetTLP focuses on prototyping new PCIe devices in software. For this purpose, manipulating TLPs with software is one of the essential functionalities. In addition, how devices and CPUs interact with each other must be designed flexibly. ExpEther does not focus on this point so that it extends PCIe links over Ethernet, and the software PCIe devices on Thunderclap pretend existing devices to reveal vulnerabilities through their drivers. In the NetTLP platform, researchers and developers can design how new software devices interact with CPUs through root complexes. More specifically, it is possible to design and implement the usage of registers of the software devices, e.g., descriptor rings, from scratch on the NetTLP platform. This functionality enables designing and implementing nonexistent devices and observing its interaction in software (Section 6.2).

3.1 Platform Overview

A key component of NetTLP delivering TLPs over Ethernet is a NetTLP adapter, which is an FPGA-based add-in card equipped with a PCIe link connected to the host and an Ethernet link. Another key component is LibTLP, which is a software library of the PCIe transaction layer on the IP network stack. The NetTLP platform is composed of two hosts, an adapter host having the NetTLP adapter and a device host where LibTLP-based applications are run, as illustrated in Figure 3.

The NetTLP adapter is responsible for the bridge between the hardware data link layer and the software transaction layer depicted in Figure 2. The NetTLP adapter delivers TLPs be-

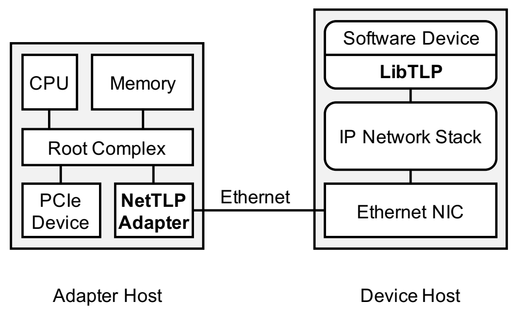


Figure 3: The overview of the NetTLP platform.

tween a host's PCIe link and the Ethernet link. When the NetTLP adapter receives TLPs from the PCIe link, the NetTLP adapter encapsulates each TLP in Ethernet, IP, UDP, and NetTLP header for sequencing and timestamping and sends the packets to the device host via the Ethernet link. When the NetTLP adapter receives a UDP packet from the Ethernet link, the NetTLP adapter checks whether the packet's payload is a TLP, decapsulates the packet, and sends the inner TLP to the PCIe link. As a result, from the perspective of the adapter host, all TLPs sent from the device host by the software are recognized as TLPs generated by the NetTLP adapter.

LibTLP implements the PCIe transaction layer in software and provides abstracted DMA APIs for applications. The applications on the device host can send and receive TLPs to the NetTLP adapter on the adapter host through UDP sockets. By using LibTLP, researchers and software developers can implement their own PCIe devices in software that perform actual behaviors of the NetTLP adapter for the root complex on the adapter host. In addition, splitting software PCIe devices and physical adapters on the distant hosts enables us to observe actual PCIe transactions flowing through the Ethernet link. We can capture the encapsulated TLPs by tcpdump at the device host or capture the TLPs on the Ethernet link by optical taps or port mirroring on Ethernet switches.

Although a NetTLP adapter is a single peripheral device, the NetTLP adapter can be applied to some PCIe communication models in PCIe topologies organized in Figure 1. Naturally, the NetTLP adapter and the software PCIe device can become a device on CPU-to-device and device-to-CPU communications. Applying the NetTLP adapter into device-to-device communication, also known as peer-to-peer DMA, realizes interactions between commercial PCIe devices and software PCIe devices. Section 6.1 shows TLPs sent from product devices by the NetTLP platform and peer-to-peer DMA integration. In addition, the NetTLP adapter can be considered a raw remote memory access device. Applications on the device host can issue DMA to any address of the memory on the adapter host through the NetTLP adapter. Section 6.3 demonstrates memory introspection methods exploiting the remote memory access by NetTLP.

3.2 TLP Processing in Software

Processing PCIe transactions in software is challenging because PCIe was originally designed to be processed by hardware. This section describes three issues to design NetTLP for achieving PCIe interactions between hardware and software.

Receiving burst TLPs: The first issue is that LibTLP needs to receive burst TLPs sent from the hardware. The minimum TLP length is 12 bytes when the TLP is a memory-read (MRd) TLP with the 32-bit address field, for instance. NetTLP adapter encapsulates TLPs with Ethernet, IP, UDP, and NetTLP headers; thus, the minimum encapsulated packet length is 64 bytes. This length is the same length as the minimum packet size of IP networks. Meanwhile, the flow control of PCIe is based on the credit system [7], and PCIe endpoints can send TLPs continuously as long as the credit remains. In particular, PCIe devices often send small TLPs using the TLP tag field to achieve high performance [21]. Because these TLP transmission intervals are continuous clock units, the throughput of encapsulated TLPs could momentarily be wire-rate on the Ethernet link with short packets.

To receive such burst TLPs by software, NetTLP exploits the TLP tag field to distribute receiving encapsulated TLPs among multiple hardware queues of an Ethernet NIC and CPU cores. The tag field is used to distinguish individual non-posted transactions that can be processed independently. The NetTLP adapter embeds the lower 4-bit of the tag values into the lower 4-bit of UDP port numbers when encapsulating TLPs. As a result, PCIe transactions to the NetTLP adapter are delivered through different UDP flows based on the tag field, and the device host can receive the flows by different NIC queues. This technique, called tag-based UDP port distribution, enables the software side to process TLPs on multiple cores associated with multiple NIC queues.

Completion Timeout: Another issue is the completion timeout. In accordance with PCIe specifications, root complexes and PCIe endpoints support a completion timeout mechanism on the PCIe transaction layer. When a requester send memory-read requests, the requester sets timeout periods for each request, and the completer need to send the completions within the periods. Hence, software PCIe devices on the NetTLP platform need to be able to send completions within the hardware-level timeout periods. The timeout period is configured in the PCIe configuration space. For instance, the minimum and maximum completion timeout periods of X520 NIC are 50 microseconds and 50 milliseconds, respectively [6]. Therefore, software PCIe devices built on LibTLP also must be capable of replying memory requests during such periods. Fortunately, Linux network stacks on general server machines are not too slow at the millisecond scale; therefore, we expect that LibTLP would meet the requirement. Section 5 examines this issue through a latency benchmark.

Encapsulation Overhead: To take TLPs to software on top of IP network stacks, NetTLP encapsulates TLPs with IP headers although encapsulation involves throughput reduction due to the header overhead. NetTLP encapsulates TLPs into multiple headers: 14-byte Ethernet, 4-byte FCS, 20-byte IP, 8-byte UDP, and 6-byte NetTLP headers. The throughput of data transfer over DMA on the NetTLP platform can be calculated with:

$$Throughput_{dma} = BW_{eth} \times \frac{TLP\_Data}{TLP\_hdr+TLP\_Data+Pkt\_Hdr+ETH\_Gap}$$

BW_{eth} is the Ethernet link speed, Pkt_Hdr is 52-byte for the headers and FCS mentioned above, and ETH_Gap is 20-byte for preamble and inter-frame gaps. From this formula, throughput with 256-byte TLP_Data (usual max payload size) and 12-byte TLP_hdr for 3DW memory-write TLP on 10 Gbps links is approximately 7.53 Gbps, which is the theoretical limitation with 10 Gbps Ethernet. Although throughput is required depending on use cases, the overhead is not significant for just prototyping software PCIe devices.

4 Implementation

We implemented the NetTLP adapter using an FPGA card and LibTLP on Linux. This section describes the details of the NetTLP adapter, APIs provided by LibTLP, hardware interrupts, and limitations of the NetTLP platform.

4.1 NetTLP Adapter

The NetTLP adapter was implemented using the Xilinx KC705 FPGA development board [51]. This board has Xilinx Kintex 7 FPGA, an Ethernet 10 Gbps port, and a PCIe Gen 2 4-lane link. We used the board because its PCIe Endpoint IP core enables user-defined logic to handle raw TLP headers. This feature is suitable for designing the NetTLP adapter. However, the Xilinx's newer PCIe IP core, which supports PCIe Gen 3, does not allow user-defined logic to handle raw TLP headers. Therefore, the current implementation of the NetTLP adapter does not support PCIe Gen 3.

Figure 4 shows the overview of the circuit diagram of the NetTLP adapter. The current NetTLP adapter has three base address register (BAR) spaces for different roles. BAR0 is used to configure the NetTLP adapter. The configurations through BAR0 support changing source and destination MAC addresses and source and destination IP addresses for encapsulating TLPs. The BAR2 space is used for the MSI-X table to support the hardware interrupts from software PCIe devices. The detail of MSI-X in NetTLP is described in Section 4.3. Both BAR0 and BAR2 memory spaces are implemented with Block RAM on the FPGA, and the NetTLP adapter has the Peripheral I/O (PIO) engine above the BAR0 and BAR2 to reply with completion TLPs for operations to the BARs.

BAR4 is different from BAR0 and BAR2; BAR4 space is connected to the Ethernet PHY and not connected to the PIO engine. All TLPs from the root complex or other devices to

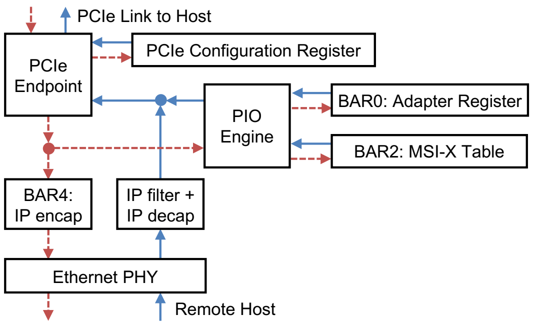


Figure 4: The circuit diagram of the NetTLP adapter.

the BAR4 space are encapsulated in Ethernet, IP, UDP, and NetTLP headers and transmitted to an external host via the Ethernet link. Namely, LibTLP on the device host communicates to the root complex on the adapter host through the memory region assigned to the BAR4.

When encapsulating TLPs to the BAR4, source and destination port numbers of the UDP headers are generated based on the tag field of their TLP headers. This is the tag-based UDP port distribution described in Section 3.2. In the current implementation, the UDP port numbers are generated with $0x3000 + (TLP\_Tag \wedge 0x0F)$. Thus, the NIC on the device host receives the TLPs by a maximum of 16 hardware queues. When the NetTLP adapter receives UDP packets from the device host, the IP filter logic checks whether the IP addresses match the configured addresses on BAR0. If the IP addresses and port numbers are correct, the packets are decapsulated, and the inner TLPs are sent to the host via the PCIe link.

The driver for the NetTLP adapter depends on types of software PCIe devices. If a software PCIe device is an Ethernet NIC, the driver is for the Ethernet NIC, and if a software PCIe device is an NVMe SSD, the driver is for the NVMe SSD. Regardless of the driver types, we implemented a simple driver that supports basic functionalities for the NetTLP adapter. This driver enables the NetTLP adapter, initializes MSI-X, and prepares a simple messaging API. The software PCIe device on the device host can obtain information about the NetTLP adapter, i.e., addresses of the BAR spaces of the adapter, PCIe bus and slot numbers, and MSI-X table. Users of the NetTLP platform can develop drivers for their software PCIe devices by extending the basic NetTLP driver.

4.2 LibTLP

LibTLP is a userspace library that implements the PCIe transaction layer. On top of the transaction layer implementation, the LibTLP provides a well-abstracted DMA API and a callback API for handling each type of TLPs.

Figure 5 shows the DMA API of LibTLP. A LibTLP instance that contains a socket descriptor, a tag value, and a destination address of a target NetTLP adapter is represented

```
ssize_t dma_read(struct nettlp *nt, uintptr_t addr, void *buf, size_t count);
ssize_t dma_write(struct nettlp *nt, uintptr_t addr, void *buf, size_t count);
```

Figure 5: The DMA API of LibTLP.

by a `nettlp` structure initialized by `nettlp_init()`. The DMA APIs for DMA reads and DMA writes are invoked by specifying a `nettlp` structure. As with the `read()` and `write()` system calls, `dma_read()` attempts to read up to `count` bytes into `buf` and `dma_write()` writes up to `count` bytes from `buf`. `addr` indicates a target address of a DMA transaction. The return values of the functions are the number of bytes read or written, or -1 and `errno` is set on error. For the DMA reads, applications can notice TLP loss or completion errors through the return value and `errno`.

In addition to the DMA API that issues DMAs to the memory on the adapter host, LibTLP provides a callback API for handling TLPs from the adapter host to the device host. The callback API allows applications to register functions for major TLP types: memory read (MRd), memory write (MWr), completion (Cpl), and completion with data (CplD). When the sockets of the `nettlp` structures receive TLPs, the registered functions are invoked for the TLPs. By using this API, the applications on the device host can respond to MRd TLPs from the root complex to the BAR4 on the NetTLP adapter by sending associated CplD TLPs, for instance.

4.3 Hardware Interrupt

For hardware interrupt, the NetTLP platform supports MSI-X, which is widely used by modern PCIe devices. MSI-X interrupt is invoked by sending a MWr TLP with particular data to a specified address from a device. The address and data for the interrupt are stored in an MSI-X table on a BAR space specified by the PCIe configuration space of the device. In other words, to send a hardware interrupt by MSI-X, it is necessary to refer to the MSI-X table on the BAR.

To achieve MSI-X on the NetTLP platform, there are two approaches: (1) placing the MSI-X table on the BAR4 and a software PCIe device on a device host holds the MSI-X table, and (2) placing the MSI-X table on other BAR spaces under the PIO engine and a software PCIe device on a device host gets the MSI-X table through other communication paths. The current implementation chooses the latter approach. The former approach does not need any other communication paths to obtain the MSI-X table from the adapter host. However, the MSI-X table is initialized by the NetTLP driver, so the software PCIe device must run on the device host before the NetTLP driver is loaded on the adapter host. Moreover, software PCIe device implementations must always be capable of maintaining the MSI-X table, even if they do not use MSI-X. These characteristics might inconvenience the development of software PCIe devices. For these reasons, we placed the MSI-X table on BAR2 under the PIO engine that is controlled by only the hardware logic of the NetTLP adapter. The software PCIe devices can obtain the content of the MSI-X table through the simple messaging API provided by the basic NetTLP driver.

4.4 tcpdump and Wireshark

To observe TLPs, we slightly modified tcpdump and implemented a Wireshark plugin. In the NetTLP platform, the encapsulated TLPs flow through the Ethernet link between the NetTLP adapter and the device host; hence, the TLPs can be easily captured by the monitoring tools of IP networking. The modified tcpdump can recognize the encapsulated TLPs and display the contents of TLPs on the popular tcpdump output. The Wireshark plugin also displays the contents of TLPs on the GUI. These tools offer researchers and developers a convenient method to observe TLPs.

4.5 Limitations

The current implementation of the NetTLP platform cannot handle the PCIe configuration space. The PCIe configuration space manages properties of the PCIe device such as Device ID, Vendor ID, and address regions of BAR spaces. The PCIe configuration space is stored in the memory of the PCIe device hardware. When the host boots up or re-scans PCIe devices, the devices use the CfgRd and CfgWr TLPs to communicate with the root complex to set up their PCIe configurations. In the current implementation of NetTLP adapter, the PCIe Endpoint IP core for Kintex 7 FPGA manages the PCIe configuration space as shown in Figure 4. The IP core does not allow user-defined logic to manipulate the configuration registers by raw TLPs. Therefore, the software PCIe devices cannot see and change their PCIe configurations. As a result, the current implementation does not support functionalities that require the manipulation of PCIe configuration registers, i.e., changing structures of MSI-X tables and SR-IOV.

5 Micro-benchmarks

To estimate performances of the software PCIe devices and applications, we conducted micro-benchmarks for the throughput and latency of DMAs on the NetTLP platform.

In the NetTLP platform, there are two directions of PCIe transactions: (1) from LibTLP to the NetTLP adapter, and (2) from the NetTLP adapter to LibTLP. They represent DMA reads and writes from a software PCIe device to a root complex, and DMA reads and writes from a root complex to a software PCIe device, respectively. In the former direction, we assume that PCIe transactions issued from the software PCIe device can be processed without packet loss because all of the components on the adapter host is hardware, which has higher bandwidth (the 16 Gbps PCIe Gen 2 4-lane link of the NetTLP adapter) than the 10 Gbps Ethernet link. In

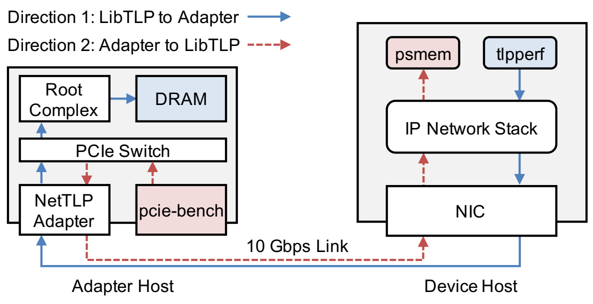


Figure 6: Benchmark setup.

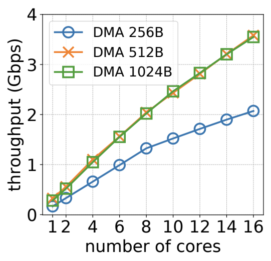


Figure 7: DMA Read throughput from LibTLP to the NetTLP adapter versus the number of CPU cores.

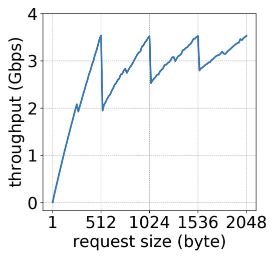


Figure 8: DMA Read throughput from LibTLP to the NetTLP adapter versus the request sizes.

the opposite direction, DMA reads from the root complex to the software PCIe devices also would not be dropped because the root complex does not send a memory read request until receiving a completion for the last read request (non-posted transaction). Based on this assumption, we measured throughput of DMA reads and writes from LibTLP (Section 5.1), and DMA reads from the NetTLP adapter (Section 5.2).

By contrast, the throughput of DMA writes from the root complex to the software PCIe device cannot be measured. The root-complex can send MWr TLPs up to the link speed of the NetTLP adapter without explicit acknowledgment (posted transactions). The current NetTLP adapter does not have mechanisms to notify congestion on the Ethernet link to the root complex; therefore, MWr TLPs are dropped if the 10 Gbps Ethernet link of the NetTLP adapter overflows. Notifying the overflow to the root complex and other devices is a future work. However, for recent peripherals such as Ethernet NICs and NVMe SSDs, usual use cases of memory writes to PCIe devices are updating registers on the devices from CPUs, and these do not require significant throughput. Therefore, we argue that the current NetTLP adapter is sufficient to prototype PCIe devices in software.

Figure 6 depicts the two directions and the components we used to generate PCIe transactions for the benchmarks. To generate PCIe transactions from LibTLP to the NetTLP adapter, we developed a LibTLP-based benchmark application called tlpperf. Users can send memory read and write requests to the memory on the adapter host through the NetTLP adapter from the device host by using tlpperf. For generating PCIe transactions in the opposite direction, we implemented a LibTLP-based pseudo memory device, called psmem, and slightly modified the pcie-bench [34]. psmem on the device host pretends a memory region associated with the BAR4 of the NetTLP adapter. As described in Section 4.1, TLPs to the BAR4 space of the NetTLP adapter are transmitted to the device host. When psmem receives a MWr TLP, psmem saves the data and the associating address. When psmem receives a MRd TLP, psmem sends CplD TLP(s) with proper data associating the requested address. In addition, to generate memory requests to the BAR4 of the NetTLP adapter, we modified pcie-bench implementation for NetFPGA-SUME. The modified pcie-bench can use the BAR4 space as the benchmark destination instead of main memory.

For the micro-benchmark, we prepared two machines for the adapter and device hosts. The adapter host was an Intel Core i9-9820X 10 core CPU and 32 GB DDR4 memory with an ASUS WS X299 SAGE motherboard. This motherboard has PCIe switches. The NetTLP adapter and the NetFPGA-SUME card for pcie-bench were installed on PCIe slots under the same PCIe switch. The device host was an Intel Core i9-7940X 12 core CPU, 32 GB DDR4 memory, and Intel X520 10 Gbps NIC with an ASUS PRIME X299-A motherboard. The device host was connected to the NetTLP adapter on the adapter host via a 10 Gbps Ethernet link. OSes were Linux kernel 4.20.2. Note that we enabled hyperthreading on the device host that had 12 physical cores to fully utilize 16 NIC queues by the tag-based UDP port distribution.

In the experiments described in this section, all the throughput results are goodput. The throughput does not include TLP and encapsulation headers. In addition, all memory requests in each iteration access the same address. We measured throughput and latency with random and sequential access patterns; however, there were no differences because of the memory access patterns in any experiment. The processing time for the software part is relatively dominant and obscures differences in performances because of the memory access patterns.

5.1 LibTLP to NetTLP Adapter

In the first benchmark, we measured the throughput of PCIe transactions from LibTLP to the memory on the adapter host through the NetTLP adapter. It is expected that the throughput would be limited by Linux kernel network stack performance where tlpperf runs because the data path on the adapter host is fully hardware and its links are the 16 Gbps PCIe Gen 2 4-lane link and the 10 Gbps Ethernet link.

Figure 7 shows the throughput of DMA reads issued by tlpperf on the device host. The key indicates request sizes of each DMA read request. As shown, the throughput linearly increases along with the number of CPU cores. This result

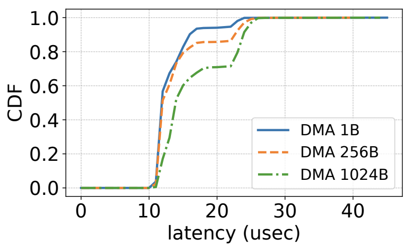


Figure 9: DMA Read latency from LibTLP to the NetTLP adapter.

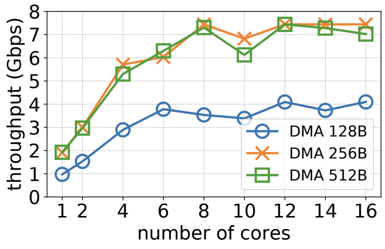


Figure 10: DMA Write throughput from LibTLP to the NetTLP adapter versus the number of CPU cores.

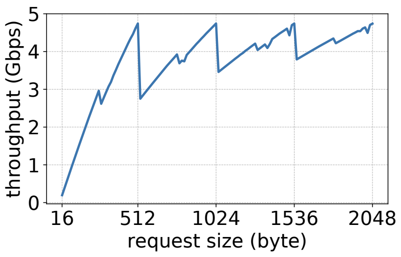


Figure 11: DMA Read throughput from the NetTLP adapter to LibTLP.

indicates that the tag-based UDP port distribution technique successfully utilizes multiple queues and multiple cores on the Linux-based device host. On the other hand, the read request size greater than 512-byte does not contribute to the throughput because the maximum read request size (MRRS) is 512-byte. The maximum throughput in this direction, which is approximately 3.6 Gbps, is less than the PCIe Gen 2 x1 link speed; however, the required throughput depends on applications and use cases. For example, Section 6 demonstrates use cases not depending on throughput. Note that the current LibTLP uses Linux Socket API; therefore, LibTLP would handle more UDP traffic with high-speed network I/O [28, 39].

Next, we measured the throughput of DMA reads from tlpperf with 16 cores while increasing the read request sizes by 16 bytes. The result shown in Figure 8 represents the *saw-tooth pattern*, which is also noted in the pcie-bench paper [34]. The saw-tooth pattern is caused by the packetized structure of the PCIe protocol. MRRS is 512-byte; therefore, when the request size is not a multiple of 512, the remaining bytes are transferred by a small size memory read. Such small-sized TLPs cause throughput reduction. Sizes of the small TLPs increase as the request sizes increase, so the throughput also increases until the request size exceeds the next multiple of 512. Slight reductions of the throughput after multiples of 256-byte are caused by the maximum payload size (MPS), which is 256-byte, in a similar manner. This result where the saw-tooth pattern appears as well as the hardware-based measurement by pcie-bench indicates that LibTLP correctly implements the packetization of the PCIe protocol.

In addition to the throughput, we measured the latency for DMA reads. The PCIe specification defines completion timeout; thus, evaluating the DMA read latency is crucial for prototyping PCIe devices in software. Figure 9 shows the result of 10000 DMA reads with 1-byte, 256-byte, and 1024-byte read requests generated by tlpperf. The latency increases with the request sizes; however, 99% latency is less than 27 microseconds regardless of the request sizes, and the maximum latency is 45 microseconds with 1024-byte DMA reads. These results correspond to the completion timeout range A (50 us to 10 ms). Therefore, we argue that prototyping PCIe devices in software with hardware root complexes is feasible from a latency perspective. According to the pcie-bench [34], DMA read latency inside a physical host is from 400 to 800 nanoseconds. Consequently, software processing for the network stack and the tlpperf application on the device host is dominant in the latency of the NetTLP platform.

We next measured the throughput of DMA writes from LibTLP. In contrast to DMA reads, DMA writes are posted transactions; therefore, we cannot measure the latency and throughput of DMA writes correctly. In this experiment, tlpperf calculates throughput when MWr TLPs are written to sockets. Figure 10 shows this measurement result. In addition to the DMA read results, DMA writes can also effectively use multiple cores and queues. In contrast to DMA reads, DMA writes reach the upper throughput with 256-byte DMA writes because MPS is 256-byte. Note that this throughput can be considered as transmitting throughput for UDP sockets of the Linux network stack.

5.2 NetTLP Adapter to LibTLP

For the second direction, we measured the throughput by generating PCIe transactions from the pcie-bench on the adapter host to the psmem running on the device host. Figure 11 shows the DMA read throughput on this direction. The result also represents the saw-tooth pattern as well as the opposite direction. Moreover, the thing that pcie-bench works with the software memory device demonstrates that the NetTLP platform can prototype one of the PCIe devices in software. Besides, the maximum throughput is approximately 4.7 Gbps. We confirmed that pcie-bench used TLP tag values from 0x00 to 0x17; thus, psmem with LibTLP could utilize the 16 cores in parallel by the tag-based UDP port distribution.

Table 2 shows DMA read latency from the pcie-bench on the adapter host to the psmem on the device host. We measured 100000 DMA reads for each request size. As shown, there are no significant differences by request sizes, unlike the original pcie-bench evaluation in hardware. This result is because the software processing on the device host—receiving and sending UDP packets—is dominant. However, the latency also meets the completion timeout range A.

Table 2: DMA Read latency from the pcie-bench to the psmem (microseconds).

Request size	Min	Median	Max	Stddev
256	14.312	17.268	87.456	1.321
512	12.2	18.764	68.552	1.550
1024	12.256	20.06	52.608	1.685
2048	11.612	18.588	68.224	2.385

6 Use Cases

This section demonstrates three use cases of NetTLP. We (1) observed specific behaviors of a commercial root complex and peripherals by capturing TLPs, (2) implemented a theoretical model of an Ethernet NIC as an actual NIC, and (3) demonstrate memory introspection for physical machines. All observations and demonstrations in this section were conducted on the same machines used in the micro-benchmarks.

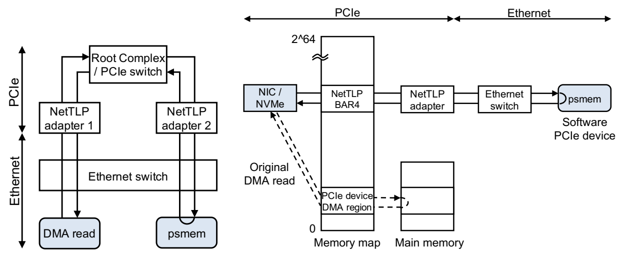
6.1 Capturing TLPs

As the first demonstration, we observe PCIe transactions of a commercial root complex, two Ethernet NICs, and two NVMe SSDs by capturing TLPs. The NetTLP adapter delivers TLPs over Ethernet links; thus, NetTLP enables us to analyze TLPs by using powerful IP network software with UNIX commands, i.e., tcpdump. Besides, the flexibility of PCIe topologies enables us to adapt NetTLP to observe various PCIe transactions issued and processed by different elements.

6.1.1 Root Complex and PCIe Switch

The first observation is to clarify the behavior of root complex. The PCIe specification does not allow PCIe switches to modify PCIe packets during switching. However, root complexes are permitted to split a PCIe packet into small PCIe packets when routing the PCIe packets between PCIe devices. The specification does not describe detailed mechanisms of TLP splitting on peer-to-peer device communication by root complexes. Although TLP splitting may negatively affect performance, its behavior depends on each root complex implementation, and observing the behavior is difficult. As a demonstration, we clarify this point by comparing actual TLPs through a root complex or a PCIe switch captured by NetTLP.

To capture the TLPs, we prepared two NetTLP adapters under the root complex or the PCIe switch on the machine used in the micro-benchmark. Figure 12a shows the topology for this observation. In the test scenario, a DMA read application on the device host sent a 512-byte MRd TLP to psmem through the two NetTLP adapters on the adapter host, and psmem returned CplD TLPs. Moreover, we switched the intermediate element from the PCIe switch to the root complex by changing PCIe slots where the NetTLP adapters were



(a) Root Complex. (b) NIC and NVMe devices.

Figure 12: Two topologies for capturing TLPs. We captured TLPs by port mirroring on the Ethernet switch.

installed. On this topology, we captured TLPs before and after passing through the PCIe switch or root complex by port mirroring on the Ethernet switch.

Figure 13 shows the captured TLPs. The x-axis indicates timestamps when a capture machine captured the TLPs from the mirror port. Note that the timestamps were stamped by NIC hardware so that the accuracy was on the nanosecond scale. The y-axis indicates TLP tag values of the TLPs. The TLPs were captured twice: before and behind the PCIe switch or root complex. The graphs on the upper row and lower row show the TLPs captured on the links connected to the NetTLP adapter 1 and adapter 2 depicted in Figure 12a, respectively.

Figure 13a confirms that the PCIe switch does not modify the TLPs as expected. The DMA read application sent a 512-byte MRd TLP, and psmem returned two 256-byte CplD TLPs. By contrast, Figure 13b reveals that the root complex split a 512-byte MRd TLP into eight 64-byte MRd TLPs with different TLP tag values when routing the TLPs between the NetTLP adapters. psmem returned eight 64-byte CplD TLPs, and the root complex rebuilt two 256-byte CplD TLPs from the small CplD TLPs. As a result, the DMA read application received the expected CplD TLPs that are aligned with MPS.

6.1.2 Ethernet NIC and NVMe SSD

Next, we measured and compared TLPs generated by commercial NIC and NVMe devices. Knowledge of how product devices use TLPs would be a useful guideline for developing PCIe devices with high performance. General peripheral devices communicate with the CPU by DMA to the main memory. To capture the TLPs from the devices, we used modified netmap drivers [39] for Ethernet NICs and a modified UNVMe [32] for NVMe SSDs to change the DMA address from main memory to the BAR4 of the NetTLP adapter. As a result, NetTLP enables capturing the TLPs sent from the devices on the Ethernet link connected to the NetTLP adapter.

For observing various behaviors of PCIe devices, we prepared different types and speeds of devices: Intel X520 and XL710 NICs, and Intel P4600 and Samsung PM1725a NVMe SSDs. Throughput of the devices are as follows: Intel X520 is

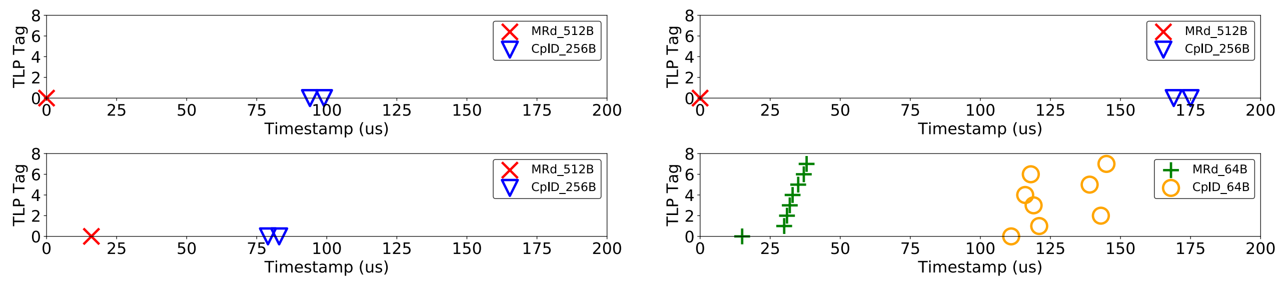
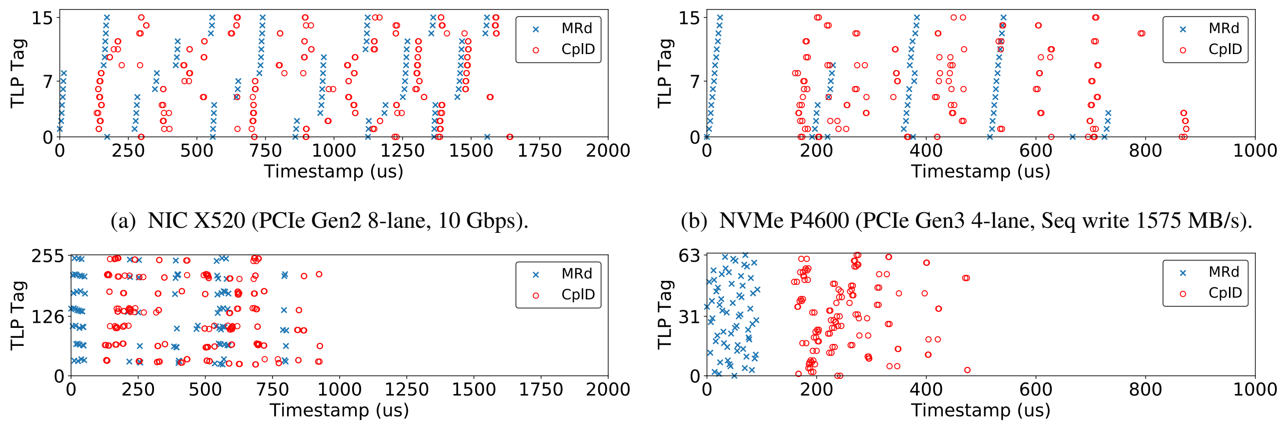

(a) A 512-byte memory read via the PCIe switch. (b) A 512-byte memory read via the root complex.

Figure 13: Comparison of captured TLPs of DMA read across the PCIe switch or the root complex. The graphs on the lower row indicate the captured TLPs behind the PCIe switch or the root complex.


(a) NIC X520 (PCIe Gen2 8-lane, 10 Gbps). (b) NVMe P4600 (PCIe Gen3 4-lane, Seq write 1575 MB/s).

(c) NIC XL710 (PCIe Gen3 8-lane, 40 Gbps). (d) NVMe PM1725a (PCIe Gen3 8-lane, Seq write 2600 MB/s).

Figure 14: Comparison of tag field usage of the NIC and NVMe devices.

a 10 Gbps Ethernet NIC, Intel XL710 is a 40 Gbps Ethernet NIC, the sequential write speed of the Intel P4600 NVMe device is 1575 MB/s, and the sequential write speed of the Samsung PM1725 is 2600 MB/s. Figure 12b shows the experimental setup of this observation. In this setup, NIC or NVMe and the NetTLP adapter were installed in PCIe slots under the same PCIe switch. The devices sent MRd TLPs for sending packets or writing data to the NVMe SSDs, and the MRd TLPs were delivered to psmem. psmem then returned CplD TLPs with Ethernet frames prepared in advance for the NIC scenario or zero-filled data for the NVMe scenario. For the NIC scenario, the NICs sent 32 1500-byte packets, and for the NVMe scenario, the NVMe SSDs wrote 32-MB data to the SSDs. Note that the block size of the Intel P4600 is 512 bytes and that of the Samsung PM1725a is 4096 bytes; therefore, we adjusted the number of NVMe write commands to write 32-MB data. To capture the TLPs, we used port mirroring on the Ethernet switch between the NetTLP adapter and the device host where psmem runs as well as the last experiment.

Figure 14 shows the result of captured TLPs of the NIC and NVMe devices. The result reveals that each PCIe device uses the TLP tag differently. X520 and P4600 use tag values from 0 to 15, PM1725a uses values from 0 to 63, and XL710 uses values from 24 to 249. The PCIe link speeds have been improved along with generations of PCIe; however, MPS has hardly improved. As a result, these PCIe devices improve data transfer throughput by sending memory requests continuously by leveraging TLP tags. The numbers of used tag values increase along with the desired throughput of the devices. According to the latency measurement in pcie-bench [34], the latency of a 512-byte DMA read is approximately 580 nanoseconds. The calculated throughput from this latency is about 7 Gbps when not using the tag field. Therefore, exploiting the tag field well is an important matter to achieve the desired throughput as this observation revealed.

6.2 Prototyping an Ethernet NIC

To confirm that NetTLP can prototype PCIe devices in software, we implemented an Ethernet NIC as a proof-of-concept on the NetTLP platform. The target NIC we implemented is simple NIC introduced by pcie-bench [34]. The original

```
23:51:48.210015 IP adapter.12291 > libtlp.12291: NetTLP: MWr, len 1, requester 00:00, tag 0x03, last 0x0, first 0xf, Addr 0xa0000010
23:51:48.210055 IP libtlp.12291 > adapter.12291: NetTLP: MRd, len 4, requester 1b:00, tag 0x03, last 0xf, first 0xf, Addr 0x2f001000
23:51:48.210069 IP adapter.12291 > libtlp.12291: NetTLP: CplD, len 4, completer 00:00, success, bc 16, requester 1b:00, tag 0x03, lowaddr 0x00
23:51:48.210083 IP libtlp.12291 > adapter.12291: NetTLP: MRd, len 25, requester 1b:00, tag 0x03, last 0x3, first 0xf, Addr 0x3bb26800
23:51:48.210095 IP adapter.12291 > libtlp.12291: NetTLP: CplD, len 25, completer 00:00, success, bc 98, requester 1b:00, tag 0x03, lowaddr 0x00
23:51:48.210122 IP libtlp.12291 > adapter.12291: NetTLP: MWr, len 1, requester 1b:00, tag 0x03, last 0x0, first 0xf, Addr 0xfee1a000
```

(a) TLPs for transmitting a 98-byte ICMP echo packet.

```
23:51:48.210134 IP libtlp.12290 > adapter.12290: NetTLP: MWr, len 25, requester 1b:00, tag 0x02, last 0x3, first 0xf, Addr 0x2f003000
23:51:48.210139 IP libtlp.12290 > adapter.12290: NetTLP: MWr, len 4, requester 1b:00, tag 0x02, last 0xf, first 0xf, Addr 0x2f002000
23:51:48.210141 IP libtlp.12290 > adapter.12290: NetTLP: MWr, len 1, requester 1b:00, tag 0x02, last 0x0, first 0xf, Addr 0xfee03000
23:51:48.212602 IP adapter.12288 > libtlp.12288: NetTLP: MWr, len 1, requester 00:00, tag 0x00, last 0x0, first 0xf, Addr 0xa0000014
23:51:48.212620 IP libtlp.12288 > adapter.12288: NetTLP: MRd, len 4, requester 1b:00, tag 0x00, last 0xf, first 0xf, Addr 0x2f002000
23:51:48.212633 IP adapter.12288 > libtlp.12288: NetTLP: CplD, len 4, completer 00:00, success, bc 16, requester 1b:00, tag 0x00, lowaddr 0x00
```

(b) TLPs for receiving a 98-byte ICMP reply packet.

Figure 15: tcpdump output of captured TLPs of the simple NIC implementation. `len` indicates length of data payload in DWORD.

simple NIC is a theoretical model of a simplistic Ethernet NIC, which does not have any performance optimizations such as DMA batching, for understanding PCIe interactions and calculating bandwidth. This nonexistent NIC model was a good target for demonstrating the productivity of NetTLP. NetTLP enables us to prototype such models of PCIe devices in software and confirm whether the models actually work with existent hardware root complexes.

The detailed interactions between a host and a simple NIC device are described in the model's implementation [2]. On the TX side, (1) the host updates a 4-byte TX queue tail pointer, (2) the device reads a 16-byte TX descriptor on the host memory, (3) the device reads the packet content and transmits the packet, and (4) the device generates 4-byte interrupt. On the RX side, (1) the host updates a 4-byte RX queue tail pointer, (2) the device reads a 16-byte RX descriptor on the host memory, (3) the device writes a received packet to the host memory, (4) the device generates 4-byte RX interrupt.

Our simple NIC implementation on the NetTLP platform performs an actual NIC with a physical root complex on the adapter host following the model's PCIe interaction. The implementation consists of two parts: a device driver for the NetTLP adapter and a software simple NIC device implementation using LibTLP. The device driver based on the basic driver treats the NetTLP adapter as an Ethernet NIC as well as usual drivers for hardware NICs. The software simple NIC creates a tap interface on the device host and uses the tap interface as its Ethernet port. The Ethernet frames transmitted to the NetTLP adapter are transferred to the device host as TLPs over the PCIe links and Ethernet links, and the Ethernet frames are transmitted to the tap interface. The software simple NIC implementation is about 400 lines of C codes, and it actually performs an Ethernet NIC.

TLPs of the simple NIC generated by the root complex and LibTLP can be observed on the Ethernet link. Figure 15a shows TLPs captured by the modified tcpdump when sending an ICMP echo packet through the simple NIC. The driver writes a TX queue pointer on the BAR4 of the NetTLP adapter (1st TLP), the simple NIC reads the TX descriptor and the packet content on 0x3bb26800 (2nd to 5th TLPs), and the simple NIC generates interrupt to 0xfee1a000 pointed by the MSI-X table after sending the packet to the tap interface (6th TLP). On the RX side shown in Figure 15b, the interaction starts from writing the received ICMP reply packet to the host memory (1st TLP) because the driver told the RX buffer to the simple NIC before receiving new packets. Afterward, the simple NIC updates the RX descriptor (2nd TLP) and generates an interrupt (3rd TLP). After the host consumes the received packet, the driver sends the buffer back to the simple NIC by updating the RX queue tail pointer (4th TLP), and the simple NIC reads the RX descriptor (5th and 6th TLPs). In this manner, the NetTLP enables implementing PCIe devices in software with hardware root complexes. Moreover, the interactions can be observed by the IP networking technique.

6.3 Physical Memory Introspection

The NetTLP provides flexible programmability for TLP interactions between hardware and software. This characteristic offers adaption of NetTLP to other use cases, for example, memory introspection. Methods for monitoring and injecting data on memory have been investigated for both physical [3,12,29,41,46] and virtual [17,49] environments. NetTLP also provides accesses to host memory via PCIe, which is similar to previous studies. However, the NetTLP adapter is a channel to manipulate the host memory remotely; therefore, researchers can implement their introspection and detection methods on top of LibTLP and IP network stack without implementing dedicated hardware or virtual machine monitors.

As the third use case, we demonstrate the possibility of adopting NetTLP into remote memory introspection through two naive applications. The first application is process-list command similar to an example of LibVMI [27]. The process-list collects process information on the Linux host equipped with a NetTLP adapter. Figure 16a shows an example usage of the process-list. When the process-list is executed, it finds

```
$ ./process-list -r 192.168.10.1 -b 1b:00 -s System.map
PhyAddr             PID STAT COMMAND
0x0000000003411740    0 R    swapper/0
0x0000087c330000      1 S    systemd
0x0000086a411dc0    647 S    systemd-journal
0x0000086ba50000    655 S    lvmetad
0x0000086c4cd940    662 S    systemd-udevd
                     ...snipped...
```

(a) process-list to collect process information.

```
$ ./codedump -r 192.168.10.1 -b 1b:00 -s System.map -p 20286 -o code.dump
code area: 0x85a48b000-0x85a48bdb0
dump complete
$ file code.dump
code.dump: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/l, missing section headers
```

(b) codedump to obtain the code area of a specified process.

Figure 16: Two example applications for physical memory introspection by NetTLP. Both applications are executed on the device host and read the physical memory of the adapter host.

an address of `task_struct` representing the first process from the specified System.map of the adapter host. Next, the process-list starts to walk through `task_struct` structures of the adapter host using `dma_read()`.

Next, let us focus on a single process. codedump obtains a binary of a running process from the adapter host. Figure 16b shows an example usage of this command. The codedump finds `task_struct` for the specified process ID by using the same methods of the process-list and obtains `mm_struct` representing the virtual memory of the process. The codedump then converts process-specific addresses for the code area into corresponding physical addresses by walking through the page table. Lastly, the dumped code area by DMA reads from LibTLP can be treated as a usual binary object file that can even be reassembled by objdump command. In these manners, researchers and developers can easily implement their memory introspection methods on the NetTLP platform.

7 Related Work

Future Interconnect: Some next-generation interconnect specifications are designed by extending the functionality of PCIe. CCIX [13] and CXL [14] introduce cache coherency between processors and peripherals to their interconnects. CCIX uses the PCIe data link layer and defines its transaction layer, and CLX defines CLX extensions on the PCIe data link layer and transaction layer. OpenCAPI [36] and Gen-Z [18] support IEEE 802.3 Ethernet and the PCIe physical layer. These interconnects require hardware extensions for both peripherals and host chipsets. Although such next-generation interconnects introduce new features, they are still packet-based data communications. NetTLP delivers TLPs over Ethernet by exploiting the packet-based communications. Thus, we believe that the NetTLP design can be applied to future interconnects as long as they adopt the layering model and packet-based communications.

Difference between NetTLP and RDMA: As with NetTLP, Remote DMA (RDMA) protocols also achieve DMA from distant hosts over Ethernet and IP networks for high speed interconnect. RoCEv2 encapsulates the Infiniband header and payload with Ethernet, IP, and UDP headers [8]. iWARP uses Ethernet, IP, and TCP headers [38]. In contrast to their purposes, NetTLP aims to provide the observability of PCIe transactions; therefore, it adopts directly encapsulating TLPs in IP and Ethernet. RDMA protocols need to convert the PCIe protocol into RDMA protocol in RDMA adapters. Thus, they lack observability of PCIe protocols that we demonstrated through the use cases.

Device drivers for software PCIe devices: NetTLP has made PCIe prototyping easier, but it has not contributed to the productivity of device drivers. Developing device drivers still requires certain effort. For improving the productivity of device drivers in the NetTLP platform, there are two approaches: the first approach is to use frameworks that automatically generate device drivers from templates related to protocol specifications and device characteristics [42,43]. Another approach is to write device drivers in userspace as with DPDK [20] while using some assists [19].

8 Conclusion

In this paper, we have proposed NetTLP that enables developing software PCIe devices that can interact with hardware root complexes. The key technique to achieve the platform is to separate the PCIe transaction layer into software and then connect the software transaction layer and the hardware data link layer by delivering TLPs over Ethernet and IP networks. Researchers and developers can prototype their own PCIe devices in software and observe actual TLPs by the IP networking techniques such as tcpdump. The use cases in this paper showed the observation of the TLP-level behaviors of the root complex and the product NICs and NVMe SSDs, the 400 LoC software Ethernet NIC implementation interacting with the hardware root complex, and physical memory introspection. We believe that the high productivity and observability on the NetTLP platform demonstrated through the use cases contribute to current and future PCIe development on both research and industrial communities.

Acknowledgments: We would like to thank the NSDI reviewers and our shepherd Andrew Moore for their insightful feedbacks. This work was supported by JSPS KAKENHI Grant Number JP18K18043.

References

[1] NetTLP. https://haeena.dev/nettlp/.

[2] pcie-bench/pcie-model. https://github.com/pcie-bench/pcie-model.

[3] The LeechCore Physical Memory Acquisition Library. https://github.com/ufrisk/LeechCore.

[4] Verilator. https://www.veripool.org/wiki/verilator.

[5] SD Express Cards with PCIe and NVMe Interfaces, 2018.

[6] Intel 82599 10 GbE Controller Datasheet: 3.1.3.2.1 Completion Timeout Period, March, 2016.

[7] PCI Express Base Specification: 2.6.1. Flow Control Rules, November 10, 2010.

[8] Infiniband Trade Association. Infiniband trade association. RoCEv2. https://cw.infinibandta.org/document/dl/7781.

[9] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[10] Brad Benton. CCIX, Gen-Z, OpenCAP: Overview & comparison. https://www.openfabrics.org/images/eventpresos/2017presentations/213_CCIXGen-Z_BBenton.pdf, March 2017.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, and et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[12] Brian D. Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digit. Investig.*, 1(1):50–60, February 2004.

[13] CCIX Consortium. Cache Coherent Interconnect for Accelerators. https://www.ccixconsortium.com.

[14] CXL Consortium. Compute Express Link. https://www.computeexpresslink.org.

[15] Scott Feldman. Rocker: switchdev prototyping vehicle. http://wiki.netfilter.org/pablo/netdev0.1/papers/Rocker-switchdev-prototyping-vehicle.pdf.

[16] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 1–14, Piscataway, NJ, USA, 2018. IEEE Press.

[17] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[18] Gen-Z Consortium. Gen-Z. https://genzconsortium.org/specifications/.

[19] Matthew P. Grosvenor. Uvnic: Rapid prototyping network interface controller device drivers. *SIGCOMM Comput. Commun. Rev.*, 42(4):307–308, August 2012.

[20] Intel. Intel Data Plane Development Kit. http://www.intel.com/go/dpdk.

[21] Intel. PCI Express High Performance Reference Design. https://www.intel.com/content/www/us/en/programmable/documentation/nik1412473924913.html.

[22] Intel. Thunderbolt Technology Community. https://thunderbolttechnology.net.

[23] Jing Qu. GEM5: PciDevice.py. http://repo.gem5.org/gem5/file/tip/src/dev/pci/PciDevice.py.

[24] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[25] Kevin Lee, Vijay Rao, and William Christie Arnold. Accelerating facebook's infrastructure with application-specific hardware. https://code.fb.com/data-center-engineering/accelerating-infrastructure/.

[26] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 395–410, Renton, WA, July 2019. USENIX Association.

[27] libvmi. Virtual Machine Introspection. http://libvmi.com/.

[28] Linux. AF_XDP: optimized for high performance packet processing. https://www.kernel.org/doc/html/latest/networking/af_xdp.html.

[29] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[30] Mellanox. BlueField SoC. https://www.mellanox.com/products/bluefield-overview/.

[31] Mentor Graphics. ModelSim. https://www.mentorg.co.jp/products/fpga/verification-simulation/modelsim/.

[32] Micron. User space nvme driver. https://github.com/MicronSSD/unvme.

[33] Netronome. Nfp-4000 flow processor. https://www.netronome.com/m/documents/PB_NFP-4000.pdf.

[34] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 327–341, New York, NY, USA, 2018. ACM.

[35] NVM Express. scalable, efficient, and industry standard. https://nvmexpress.org.

[36] OpenCAPI Consortium. OpenCAPI 4.0 transaction layer specification. https://opencapi.org.

[37] OpenChannelSSD. The LightNVM qemu implementation, based on NVMe. https://github.com/OpenChannelSSD/qemu-nvme.

[38] Renato J. Recio, Paul R. Culley, Dave Garcia, Bernard Metzler, and Jeff Hilland. A Remote Direct Memory Access Protocol Specification. RFC 5040, October 2007.

[39] Luigi Rizzo. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[40] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.

[41] Joanna Rutkowska. Beyond the cpu: Defeating hardware based ram acquisition. Proceedings of BlackHat DC, 2007.

[42] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 275–288, New York, NY, USA, 2009. Association for Computing Machinery.

[43] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 73–86, New York, NY, USA, 2009. Association for Computing Machinery.

[44] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, Berkeley, CA, USA, 2014. USENIX Association.

[45] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. Cosmos openssd: A pcie-based open source ssd platform. *Proc. Flash Memory Summit*, 2014.

[46] Chad Spensky, Hongyi Hu, and Kevin Leach. Lo-phi: Low-observable physical host instrumentation for malware analysis. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[47] J. Suzuki, Y. Hidaka, J. Higuchi, T. Yoshikawa, and A. Iwata. Expressether - ethernet-based virtualization technology for reconfigurable hardware platform. In *14th IEEE Symposium on High-Performance Interconnects (HOTI'06)*, pages 45–51, Aug 2006.

[48] Jun Suzuki, Teruyuki Baba, Yoichi Hidaka, Junichi Higuchi, Nobuharu Kami, Satoshi Uchida, Masahiko Takahashi, Tomoyoshi Sugawara, and Takashi Yoshikawa. Adaptive memory system over ethernet. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[49] Gary Wang, Zachary J. Estrada, Cuong Pham, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Hypervisor introspection: A technique for evading passive virtual machine monitoring. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT'15, pages 12–12, Berkeley, CA, USA, 2015. USENIX Association.

[50] P. Willmann, Hyong-youb Kim, S. Rixner, and V. S. Pai. An efficient programmable 10 gigabit ethernet network interface card. In *11th International Symposium on High-Performance Computer Architecture*, pages 96–107, Feb 2005.

[51] Xilinx. Xilinx kintex-7 fpga kc705 evaluation kit. https://japan.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html.

[52] Noa Zilberman, Yury Audzevich, Georgina Kalogeridou, Neelakandan Manihatty-Bojan, Jingyun Zhang, and Andrew Moore. Netfpga: Rapid prototyping of networking devices in open source. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 363–364, New York, NY, USA, 2015. ACM.

Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage

Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha
University of Michigan

Abstract— **By replicating data across sites in multiple geographic regions, web services can maximize availability and minimize latency for their users. However, when sacrificing data consistency is not an option, we show that service providers have to today incur significantly higher cost to meet desired latency goals than the lowest cost theoretically feasible. We show that the key to addressing this sub-optimality is to 1) allow for erasure coding, not just replication, of data across data centers, and 2) mitigate the resultant increase in read and write latencies by rethinking how to enable consensus across the wide-area network. Our extensive evaluation mimicking web service deployments on the Azure cloud service shows that we enable near-optimal latency versus cost tradeoffs.**

1 Introduction

Replicating data across data centers is important for a web service to tolerate the unavailability of some data centers [1] and to serve users with low latency [5]. A front-end web server close to a user can serve the user's requests by accessing nearby copies of relevant data (see Figure 1). Even in collaborative services such as Google Docs and ShareLaTeX, accessing a majority of replicas suffices for a front-end to read or update shared data while preserving consistency.

However, it is challenging to keep data spread across the globe strongly consistent as no single design can simultaneously minimize read latency, write latency, and cost.

- To preserve consistency, *any* subset of sites which are accessed to serve a read must overlap with *all* subsets used for writes. Therefore, allowing a front-end to read from nearby data sites forces other front-ends to write to distant data sites, thus increasing write latency.
- Similarly, providing low read latency requires having at least one data site near each front-end, thereby increasing the total number of data sites. This inflates expenses incurred both for storage and for data transfers to synchronize data sites.

Given these tradeoffs, service providers must determine how to meet their desired latency goals at minimum cost. Or, correspondingly, how to minimize read and write latencies given a cost budget? In this paper, we make the following contributions towards addressing these questions.

1. We show that existing solutions for enabling strongly consistent distributed storage are far from optimal in trading off latency versus cost. The cost necessary to satisfy bounds on read and write latencies is often significantly higher than the lowest cost theoretically feasible. For example, across a range of access patterns and latency bounds, the state-of-the-art geo-replication protocol EPaxos [51] imposes on average 30% higher storage cost than is optimal (§5.1.2). This sub-optimality also inflates the minimum latency bounds satisfiable within a cost budget.

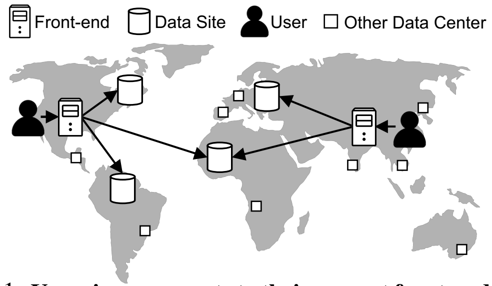


Figure 1: **Users issue requests to their nearest front-end servers which in turn access geo-distributed storage.**

2. We demonstrate the feasibility of achieving near-optimal latency versus cost tradeoffs in strongly consistent geo-distributed storage. In other words, we do not merely improve upon the status quo, but show that there remains little room for improvement over the tradeoffs enabled by PANDO, our new approach for consensus across the wide-area network. PANDO exploits the property that, from any data center's perspective, some data centers are more proximate than others in a geo-distributed deployment. Therefore, beyond reducing the *number of round-trips* of wide-area communication when executing reads and writes (as has typically been the goal in prior work [49, 42, 51]), it is equally important to reduce the *magnitude of delay incurred on every round-trip*. We apply this principle in two ways.

2a. We show how to erasure-code objects across data sites without reads incurring higher wide-area latencies compared to replicated data. By splitting each object's data and storing one split (instead of one replica) per data site, a service can use its cost budget to spread each object's data across more data centers than is feasible with replication. To leverage this increased geographic spread for minimizing latencies, PANDO separates out two typically intertwined aspects of consensus: discovering whether the last write completed, and determining how to resolve any associated uncertainty. Since writes seldom fail in typical web service deployments, we enable a client to read an object by first communicating with a small subset of nearby data sites; only in the rare case when it is uncertain whether the last write completed does the client incur a latency penalty to discover how to resolve the uncertainty.

2b. In the wide-area setting, we show how to reach consensus in two rounds, yet approximate a one-round protocol's latency. Executing writes in two rounds simplifies compatibility with erasure-coded data, and we ensure that this approach has little impact on latency. First, PANDO requires clients to contact a smaller, more proximate subset of data sites in the first round than in the second round. Second, after a client initiates the first round, it delegates initiation of the second round to a more central data center, which receives all responses from the first round. By combining these two measures, messaging delays incurred in the first phase of a write help reduce the latency incurred in the second phase, instead of adding to it.

3. We compare PANDO to state-of-the-art consensus protocols via extensive measurement-driven analyses and in deployments on Azure. In the latency–cost tradeoff space, we find that PANDO reduces by 88% the median gap between achievable tradeoffs and the best theoretically feasible tradeoffs. Moreover, PANDO can cut dollar costs to meet the same latency goals by 46% and lower 95^{th} percentile read latency by up to 62% at the same storage overhead.

2 Setting and Motivation

We begin by describing our target setting, the approach we use for enabling globally consistent reads and writes, and the shortcomings of existing solutions that use this approach.

2.1 System model, goals, and assumptions

We seek to meet the storage needs of globally deployed applications, such as Google Docs [4] and ShareLaTeX [8], in which low latency and high availability are critical, yet weak data consistency (such as eventual or causal) is not an option. In particular, we focus on enabling a geo-distributed object/key-value store which a service's front-end servers read from and write to when serving requests from users. We aim to support GETs and conditional-PUTs on any individual key; we defer support for multi-key transactions to future work. In contrast to PUTs (which blindly overwrite the value for a key), conditional-PUTs attempt to write to a specific version of a key and can succeed only if that version does not already have a committed value. This is essential in services such as Google Docs and ShareLaTeX to ensure that a client cannot overwrite an update that it has not seen.

In enabling such a geo-distributed key-value store, we are guided by the following objectives:

- **Strong consistency:** Ensure all reads and writes on any key are linearizable; i.e., all writes are totally ordered and every read returns the last successful write.
- **Low latency:** Satisfy service provider's SLOs[1] (service-level objectives) for bounds on read and write latencies, so as to ensure a minimum quality-of-service for all users. We focus on the wide-area latency incurred when serving reads or writes, assuming appropriate capacity planning and load balancing to bound queuing delays.
- **Low cost:** Minimize cost (sum of dollar costs for storage, data transfers, storage operations, and compute) necessary to satisfy latency goals. Since cost for storage operations and data transfers grows with more copies stored, in parts of the paper, we use storage overhead (i.e., number of copies stored of every data item) as a proxy for cost. This frees us from making any assumptions about pricing policy or the workload (e.g., read-to-write ratio).
- **Fault-tolerance:** Serve requests on any key as long as fewer than f data centers are unavailable.

[1]Unlike SLAs, violations of SLOs are acceptable, but need to be minimized.

We focus on satisfying input latency bounds in the absence of conflicts and failures—both of which occur rarely in practice [11, 2, 48, 29]—but seek to minimize performance degradation when they do occur (§3.5 and §5.1.2). In addition, we build upon state-of-the-art cloud services which offer low latency variance between their data centers [34] and within their intra-data center storage services (e.g., Azure's CosmosDB provides a 10 ms tail read latency SLA [12]).

Note that, in order to satisfy desired latency SLOs at minimum cost (or to minimize latencies given a cost budget), a service cannot select the data sites for an object at random. Instead, as we describe later in Section 4, any service must utilize its knowledge of an object's workload (e.g., locations of the users among whom the object is shared) in doing so.

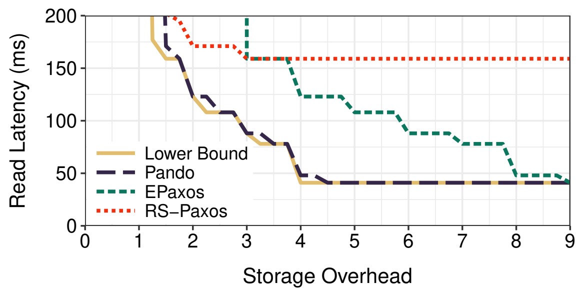
2.2 Approach

One can ensure linearizability in distributed storage by serializing all writes through a leader and rely on it for reads, e.g., primary-backup [18], chain replication [68], and Raft [57]. A single leader, however, cannot be close to all front-ends across the globe. Front-ends which are distant from the leader will have to suffer high latencies.

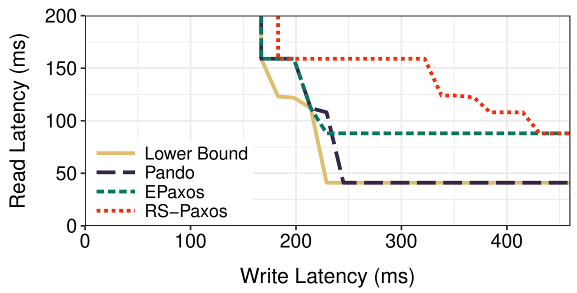
To reduce the need to contact a distant leader, one could use read leases [21, 52] and migrate the leader based on the current workload, e.g., choose as the leader the replica closest to the front-end currently issuing reads and writes. However, unless the workload exhibits very high locality, tail latency will be dominated by the latency overheads incurred during leader migration and lease acquisition.

To keep read and write latencies within specified bounds irrespective of the level of locality, we pursue a leaderless approach. Among the leaderless protocols which allow every front-end to read and write data from a subset of nearby data sites (a read or write quorum), we consider those based on Paxos because it enables consensus. Other quorum-based approaches [20] which only enable atomic register semantics (i.e., PUT and GET) are incapable of supporting conditional updates [35]. While there exist many variants of Paxos, in all cases, we can optimize latencies in two ways.

First, instead of executing Paxos, a front-end can read an object by simply fetching the object's data from a read quorum. To enable this, a successful writer asynchronously marks the version it wrote as committed at all data sites. In the common case, when there are no failures or conflicts, a read is complete in one round trip if the highest version seen across a read quorum is marked as committed [44].



(a) Write latency ≤ 300 ms



(b) Storage overhead $\leq 6\times$

Figure 2: **Slices of the three-dimensional tradeoff space where we compare latency estimates for replication-based EPaxos [51], erasure coding-based RS-Paxos [53], and our solution PANDO against a lower bound. Front-ends are in Azure's Australia East, Central India, East Asia, East US, and Korea South data centers, whereas data sites are chosen from all Azure data centers.**

Second, instead of every front-end itself executing reads and writes, we allow for it to relay its operations through a delegate in another data center. The flexibility of utilizing a delegate can be leveraged to reduce latency when, compared to the front-end, that delegate is more centrally placed relative to the data sites of the object being accessed.

2.3 Sub-optimality of existing solutions

The state-of-the-art Paxos variant for geo-replicated data is EPaxos [51], as we show in Section 5. For typical replication factors (i.e., 3 or 5), EPaxos enables any front-end to read/write with one round of wide-area communication with the nearest majority of replicas. If lower read latencies than feasible with $2f+1$ replicas are desired, then one can use a higher replication factor N, set the size R of read quorums to be $\geq f+1$ (to ensure overlap with write quorums even in the face of f failures) and set the size W of write quorums to $N-R+1$ (to preserve consistency).

Figure 2 shows the tradeoffs enabled by EPaxos for an example access pattern. For each read latency bound, these graphs respectively plot the minimum storage overhead and write latency bounds that are satisfiable. As we discuss later in the paper, we compute these bounds by solving protocol-specific mixed integer programs (§4) which take as input the expected access pattern and latency measurements between all pairs of data centers (§5.1). We show two two-dimensional slices of the three-dimensional read latency–write latency–storage overhead tradeoff space.

To gauge the optimality of the tradeoffs achievable with EPaxos, we compare it against a lower bound. Given a bound on read latency, the minimum storage overhead necessary and the minimum write latency bound that can be satisfied cannot be lower than those determined by our lower bound. Though the lower bound may be unachievable by any existing consensus protocol, we compute it by solving a mixed integer program which assumes that reads and writes can be executed in a single round and enforces the following properties that *any* quorum-based approach must respect:

- *Tolerate unavailability of $\leq f$ data centers:* All data sites in at least one read and one write quorum must be available in the event that $\leq f$ data centers fail.
- *Prevent data loss:* At least one copy of data must remain in any write quorum when any f data sites are unavailable.
- *Serve reads:* The data sites in any read quorum must collectively contain at least one copy of the object.
- *Preserve strong consistency:* All read–write and write–write quorum pairs must have a non-empty intersection.

Equally important are constraints that we do *not* impose: all read quorums (same for write quorums) need not be of the same size, and an arbitrary fraction of an object's data can be stored at any data site.

Figure 2 shows that EPaxos is sub-optimal in two ways. First, to meet any particular bound on read latency, EPaxos imposes a significant cost overhead; in Figure 2(a), EPaxos requires at least 9 replicas to satisfy the lowest feasible read latency bound (40 ms), whereas the lower bound storage overhead is 4x. Recall that, greater the number of copies of data stored, higher the data transfer costs when reading and writing. Second, given a cost budget, the read latencies achievable with EPaxos are significantly higher than the lower bound; in Figure 2(b), where storage overhead is capped at 6x, we see that the minimum read latency achievable with EPaxos (80 ms) is twice the lower bound (40 ms).

Of course, a lower bound is just that; some of the tradeoffs that it deems feasible may potentially be unachievable. However, for the example in Figure 2 and across a wide range of configurations in Section 5, we show that PANDO comes close to matching the lower bound. We describe how next.

3 Design

The fundamental source of EPaxos's sub-optimality in trading off cost and latency is its reliance on replication. Replication-based approaches inflate the cost necessary to meet read latency goals because spreading an object's data across more sites entails storing an additional *full copy* at each of these sites. To enable latency versus cost tradeoffs that are closer to optimal, the key is to store a *portion* of an object's data at each data site, like in the lower bound.

Therefore, we leverage erasure coding, a data-agnostic approach which enables such flexible data placement while matching replication's fault-tolerance at lower cost [69]. For example, to tolerate $f = 1$ failures, instead of requiring at least $2f + 1 = 3$ replicas, one could use Reed-Solomon coding [60] to partition an object into $k = 2$ splits, generate $r = 2$ parity splits, and store one split each at $k + r = 4$ sites; any k splits suffice to reconstruct the object's data. Compared to replication, this reduces storage overhead to $2\times$, thus also reducing the number of copies of data transferred over the wide-area when reading or writing.

State-of-the-art implementations of erasure coding [9] require only hundreds of nanoseconds to encode or decode kilobyte-sized objects. This latency is negligible compared to wide-area latencies, which range from tens to hundreds of milliseconds. Moreover, the computational costs for encoding and decoding pale in comparison to costs for data transfers and storage operations (§ 5.1.3).

3.1 Impact of erasure coding on wide-area latency

While there exist a number of protocols which preserve linearizability on erasure-coded data [15, 24], they largely focus on supporting PUT/GET semantics. To support conditional updates, we consider how to enable consensus on erasure-coded data with a leaderless approach such as Paxos. We have one of two options.

One approach would be to extend one of several one-round variants of Paxos to work on erasure-coded data. However, most of these protocols require large quorums (e.g., a write would have to be applied to a super-majority [42] or even all [49] data sites), rendering them significantly worse than the lower bound. Whereas, extending EPaxos [51], which requires small quorums despite needing a single round, to be compatible with erasure-coded data is far from trivial given the complex mechanisms that it employs for failure recovery.

Therefore, we build upon the classic two-phase version of Paxos [40] and address associated latency overheads. In either phase, a writer (a front-end or its delegate) communicates with all the data sites of an object and waits for responses from a write quorum. In Phase 1, the writer discovers whether there already is a value for the version it is attempting to write and attempts to elect itself leader for this version. In Phase 2, it sends its write to all data sites. A write to a version succeeds only if, prior to its completion of both phases, no other writer has been elected the leader. If the leader fails during Phase 2 but the write succeeds at a quorum of data sites, subsequent leaders will adopt the existing value and use it as part of their Phase 2, ensuring that the value for any specific version never changes once chosen.

This natural application of Paxos on erasure-coded data, called RS-Paxos [53], is inefficient in three ways.

- **Two rounds of wide-area communication.** Any reduction in read latency achieved by enabling every front-end to read from a more proximate read quorum has twice the adverse effect on write latency. In Figure 2(b), we see that when the read latency bound is stringent (e.g., $\leq$ 100 ms), the minimum write latency bound satisfiable with RS-Paxos is twice that achievable with EPaxos. When the read latency bound is loose (e.g., $\geq$ 150 ms), write latency inflation with RS-Paxos is lower because the data sites are close to each other and front-ends benefit from delegation.

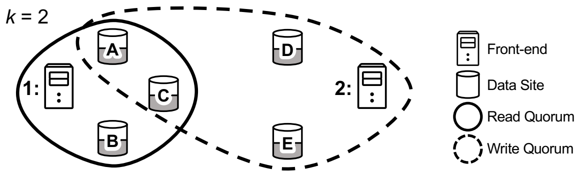


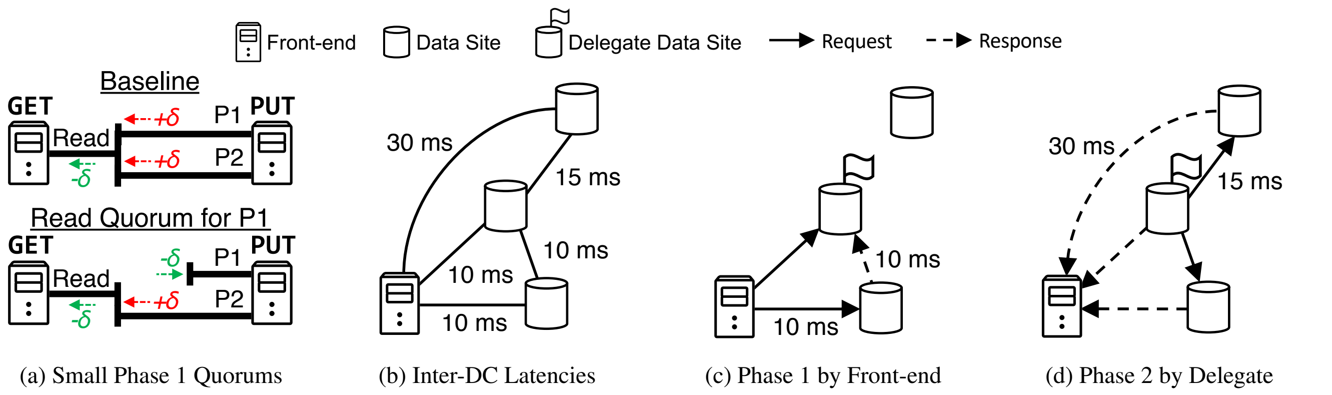
Figure 3: **Example execution of RS-Paxos on an erasure-coded object, whose data is partitioned into $k = 2$ splits. For all readers and writers to be able to reconstruct the last successful write, any write quorum must have an overlap of k or more data sites with every read and every write quorum.**

- **Increased impact of conflicts.** Executing writes in two rounds makes them more prone to performance degradation when conflicts arise. When multiple writes to the same key execute concurrently, none of the writes may succeed within two rounds. Either round of each write may fail at more than a quorum of data sites if other writes complete one of their rounds at those sites.
- **Larger intersections between quorums.** As we see in Figure 2(a), at storage overheads of 4x or more, the minimum read latency bound satisfiable with RS-Paxos is significantly higher than that achievable with EPaxos. This arises because, when an object's data is partitioned into k splits, every read quorum must have an overlap of at least k sites with every write quorum (see Figure 3). Thus, erasure coding's utility in helping spread an object's data across more sites (than feasible with replication for the same storage overhead) is nullified.

3.2 Overview of PANDO

What if these inefficiencies did not exist when executing Paxos on erasure-coded data? To identify the latency versus cost tradeoffs that would be achievable in this case, we consider a hypothetical ideal execution of Paxos on erasure-coded data: one which requires a single round of communication and can make do with an overlap of only one site between read–write and write–write quorum pairs. For the example used in Figure 2, this hypothetical ideal (not shown in the figure) comes close to matching the lower bound.

Encouraged by this promising result, we design PANDO to approximate this ideal execution of Paxos on erasure-coded data. First, we describe how to execute Paxos in two rounds on geo-distributed data, yet come close to matching the messaging delays incurred with one-round protocols. Second, leveraging the rarity of conflicts and failures in typical web service workloads, we describe how to make do with a single data site overlap between quorums in the common case. Finally, we discuss how to minimize performance degradation when conflicts do arise. In our description, we assume an object's data is partitioned into k splits.



(a) Small Phase 1 Quorums (b) Inter-DC Latencies (c) Phase 1 by Front-end (d) Phase 2 by Delegate

Figure 4: **(a) Reusing read quorums in Phase 1 of writes enables reduction in read latency without impacting (Phase 1 + Phase 2) latency for writes. (b) Example deployment with one-way delays between relevant pairs of data centers shown. Phase 1 quorum size is 2 and Phase 2 quorum size is 3. If same (Phase 2) quorum were used in both phases of a write, like in RS-Paxos, write latency would be 120 ms. (c) and (d) By directing Phase 1 responses to a delegate and having it initiate Phase 2, Pando reduces write latency to 65 ms (20 ms in Phase 1 + 45 ms in Phase 2), close to the 60 ms latency feasible with one-round writes.**

3.3 Mitigating write latency

We reduce the latency overhead of executing Paxos in two rounds by revisiting the idea of delegation (§ 2.2): a front-end sends its write request to a stateless delegate, which executes Paxos and returns the response. When data sites are spread out (to enable low read latencies), two round-trips to a write quorum incurs comparable delay from the front-end versus from the delegate. The round-trip from the front-end to the delegate proves to be an overhead.

To mitigate this overhead, what if 1) transmission of the message from the front-end to the delegate overlaps with Phase 1 of Paxos, and 2) transmission of the response back overlaps with Phase 2? The latency for a front-end to execute the two-phase version of Paxos would then be roughly equivalent to one round-trip between the front-end and the delegate, thus matching the latency feasible with a one-round protocol. We show how to make this feasible in two steps.

3.3.1 Shrinking Phase 1 quorums

First, we revisit the property of classic Paxos that a writer needs responses from the same number of data sites in both phases of Paxos: the size of a write quorum. To ensure that a writer discovers any previously committed value, Paxos only requires that any Phase 1 quorum intersect with every Phase 2 quorum; Phase 1 quorums need not overlap [37]. In Pando, we take advantage of this freedom to use a smaller quorum in the first phase of Paxos than in the second phase.

We observe that the intersection requirements imposed on Phase 1 and Phase 2 quorums are precisely the properties required of read and write quorums: any read quorum must intersect with every write quorum, whereas no overlap between read quorums is required. Therefore, when executing Phase 1 of Paxos to write to an object, it suffices to get responses from a read quorum, thus allowing improvements in read latency to also benefit leader election. A writer (a front-end or its delegate) needs responses from a write quorum only when executing Phase 2.

Figure 4(a) illustrates the corresponding improvements in write latency. When a quorum of the same size is used in both phases of a write, a reduction of δ in the read latency bound results in a 2δ increase in the minimum satisfiable write latency bound (because of the need for read and write quorums to overlap). In contrast, our reuse of read quorums in the Phase 1 of writes ensures that spreading out data sites to enable lower read latencies has (roughly speaking) no impact on write latency; when read quorums are shrunk to reduce the read latency bound by δ, the increase of δ in Phase 2 latency (to preserve overlap between quorums) is offset by the decrease of δ in Phase 1 latency.

3.3.2 Partially delegating write logic

While our reuse of read quorums in Phase 1 of a write helps reduce write latency, Phase 2 latency remains comparable to a one-round write protocol. Therefore, the total write latency remains significantly higher than that feasible with one-round protocols.

Pando addresses this problem via *partial* use of delegation. Rather than having a front-end executing a write either do all the work of executing Paxos itself or offload all of this work to a delegate, we offload *some* of it to a delegate.

Figures 4(c–d) show how this works in Pando. A front-end initiates Phase 1 of Paxos by sending requests to data sites of the object it is writing to, asking them to send their responses to a chosen delegate. In parallel, the front-end sends the value it wants to write directly to the delegate. Once the delegate receives enough responses (i.e., the size of a read quorum), it will either inform the front-end that Phase 1 failed (the rare case) or initiate Phase 2 (the common case), sending the value to be written to all data sites for the object. Those data sites in turn send their responses directly back to the front-end, which considers the write complete once it receives responses from a write quorum.

Note that partial delegation preserves Paxos's fault tolerance guarantees. To see why, consider the case where a end-user's client sends the same request to two front-ends—perhaps due to suspecting that the first front-end has failed—and both front-ends execute the request. Paxos guarantees that at most one of these writes will succeed. Similarly, with partial delegation, in the rare case when the front-end suspects that the delegate is unavailable, it can simply re-execute both phases on its own. Paxos will resolve any conflicts and at most one of the two writes (one executed via the delegate and the other executed by the front-end) will succeed.

Thanks to the heterogeneity of latencies across different pairs of data centers, the use of small Phase 1 quorums combined with the delegation of Phase 2 eliminates most of the latency overhead of two-phase writes. In Figure 4(b-d), the two techniques reduce write latency down from 120 ms with classic Paxos to 65 ms with PANDO, only 5 ms higher than what can be achieved with a one-round protocol. The remaining overhead results from the fact that there still has to be some point of convergence between the two phases.

3.4 Enabling smaller quorums

The techniques we have described thus far lower the minimum write latency SLO that is satisfiable given an SLO for read latency. However, as we have seen in Figure 2(a), erasure coding inflates the minimum read latency SLO achievable given a cost budget (e.g., a bound on storage overhead). As discussed earlier in Section 3.1, this is due to the need for larger intersections between quorums when data is erasure-coded, as compared to when replicated.

Recall that the need for an intersection of k data sites between any pair of read and write quorums exists so that any read on an object will be able to reconstruct the last value written; at least k splits written during the last successful write will be part of any read quorum. Thus, linearizability is preserved even in the worst case when a write completes at the minimum number of data sites necessary to be successful: a write quorum. However, since concurrent writes are uncommon [48, 29] and data sites are rarely unavailable in typical cloud deployments [11, 2], most writes will be applied to all data sites. Therefore, in the common case, all data sites in any read quorum will reflect the latest write.

In PANDO, we leverage this distinction between the common case and the worst case to optimize read latency (and equivalently any write's Phase 1 latency, given that PANDO uses the same quorum size in both cases) as follows.

Read from smaller quorum in the common case. After issuing read requests to all data sites, a reader initially waits for responses from a subset which is 1) at least of size k and 2) has an intersection of at least one site with every write quorum; we refer to this as a Phase 1a quorum. In the common case, all k splits have the same version and at least one of them is marked committed; the read is complete in this case. An overlap of only one site with every write quorum

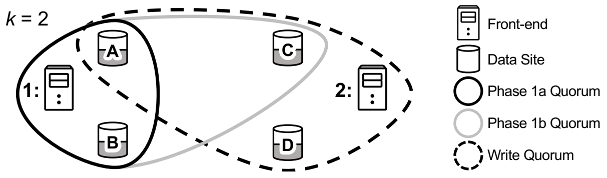


Figure 5: **For an object partitioned into $k = 2$ splits, PANDO requires an overlap of only one site between any Phase 1a and Phase 2 quorum. Responses from the larger Phase 1b quorum are needed only in the case of failure or conflict.**

suffices for the reader to discover the latest version of the object; at least one of the splits received so far by the reader will be one written by the last successful write to this object.

Read from larger quorum if failure or conflict. At this juncture, if the last successful write has not yet been applied to all data sites, the reader may only know the latest version of the object but not the value of that version. To reconstruct that value, the reader must wait for responses from more data sites until the subset it has heard from has an overlap of k sites or more with every write quorum; this is a Phase 1b quorum. As a result, a reader must incur the latency penalty of waiting for responses from farther data sites only if the last successful write was executed when either some data sites were unavailable or a conflicting write was in progress.

In the example in Figure 5, Front-end 1 can complete reading based on responses from sites A and B in the common case since two splits suffice to reconstruct the object. If the last write was from Front-end 2 and this write completed only at a subset of sites, there are two cases to consider:

- If Front-end 2's write has been applied to a write quorum (say A, C, and D), then the response from site A will help Front-end 1 discover the existence of this write. Front-end 1 needs an additional response from C in this case to be able to reconstruct the value written by Front-end 2.
- If Front-end 2's write has been applied to less than a write quorum (say, A and D), then Front-end 1 may be unable to find k splits for this version even from a Phase 1b quorum (A, B, and C). In this case, that value could not have been committed to *any* Phase 2 quorum. Therefore, the reader falls back to the previous version. PANDO garbage collects the value for a version only once a value has been committed for the next version (§4). The overhead of storing multiple versions of a key will be short-lived in our target setting where failures and write conflicts are rare.

Phase 1a and 1b quorums can also be used as described above during the first round of a write. The only difference in the case of writes is that responses from data sites can be potentially directed to a delegate at a different data center than the one which initiates Phase 1.

To preserve correctness of both reads and writes, the minimum size of Phase 1a quorums must be $\max(k, f+1)$, and Phase 1b and Phase 2 quorums must contain at least $f+k$ data sites. These quorum sizes are inter-dependent because any Phase 1a quorum must have a non-empty overlap with every Phase 2 quorum and any Phase 1b or Phase 2 quorum must have an overlap of at least k sites with every Phase 2 quorum. For each of the three quorum types, all quorums of that type are of the same size and any subset of data sites of that size represent a valid quorum of that type.

Note that, if further reductions in common-case read latency are desired, one could use timed read leases as follows [21, 52]. Instead of using the normal read path, a front-end that holds a lease for a key could cache the value or fetch it from k nearby data sites to avoid the latency of communicating with a complete Phase 1a quorum. However, this approach would not benefit tail latency for reads and may increase latency for writes.

3.5 Reducing impact of conflicting writes

Lastly, we discuss how PANDO mitigates performance degradation when conflicts arise. As mentioned before (§2), since conflicts rarely occur in practice [48, 29], we allow for violations of input latency bounds when multiple writes to a key execute concurrently. However, we ensure that the latency of concurrent writes is not arbitrarily degraded.

Our high-level idea is to select one of every key's data sites as the leader and to make use of this leader *only when conflicts arise*. PANDO's leaderless approach helps satisfy lower latency bounds by eliminating the need for any front-end to contact a potentially distant leader. However, when a front-end's attempted write fails and it is uncertain whether a value has already been committed for this version, the front-end forwards its write to the leader. In contrast to the front-end retrying the write on its own, relying on the leader can ensure that the write completes within at most two rounds.

To make this work, we ensure that any write executed by a key's leader always supersedes writes to that key being attempted in parallel by front-ends. For this, we exploit the fact that front-ends always retry writes via the leader, i.e., any front-end will attempt to directly execute a write at most once. Therefore, when executing Paxos, we permit any front-end to use proposal numbers of the form (0, front-end's ID) but only allow the leader to set the first component to values greater than or equal to 1, so that its writes take precedence at every data site.

Note that, since we consider it okay to violate the write latency bound in the rare cases when conflicts occur, we do not require the leader to be close to any specific front-end. Therefore, leader election can happen in the background (using any of a number of approaches [23, 14]) whenever the current one fails. If conflicting writes are attempted precisely when the leader is unavailable, these writes will block until a new leader is elected. Like prior work [51, 40], PANDO cannot bound worst-case write latency when conflicts *and* data center failures occur simultaneously.

A proof of PANDO's correctness and a TLA+ specification are in Appendices A and B.

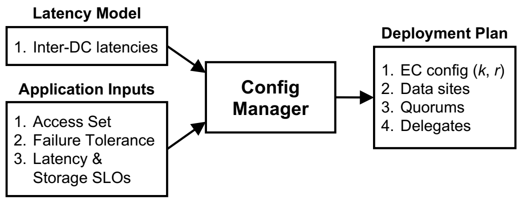


Figure 6: **Selecting a deployment plan with ConfigManager.**

4 Implementation

To empirically compare the manner in which different consensus approaches trade off read latency against write latency and cost, we implemented a key-value store which optimizes the selection of data sites for an object based on knowledge of how the object will be accessed.

ConfigManager. Central to this key-value store is the ConfigManager, which sits off the data path (thus not blocking reads and writes) and identifies *deployment plans*, one per access pattern. As shown in Figure 6, a deployment plan determines the number of splits k that the key's value is partitioned into, the number of redundant splits r, and the $k + r$ data sites at which these splits are stored; $k = 1$ corresponds to replication, and Reed-Solomon coding [60] is used when $k > 1$. The deployment plan also specifies the sizes of different quorum types and the choice of delegates (if any).

To make this determination, in addition to the application's latency, cost, and fault-tolerance goals, ConfigManager relies on the application to specify every key's *access set*: data centers from which front-ends are expected to issue requests for the key. An application can determine an object's access set based on its knowledge of the set of users who will access that object, e.g., in Google Docs, the access set for a document is the set of data centers from which the service will serve users sharing the document. When uncertain (e.g., for a public document), the access set can be specified as comprising all data centers hosting its front-ends; this uncertainty will translate to higher latencies and cost.

The ConfigManager selects deployment plans by solving a mixed integer program, which accounts for the particular consensus approach being used. For example, PANDO's ConfigManager selects a delegate and preferred quorums per front-end, using RTT measurements to predict latencies incurred. Given bounds on any two dimensions of the trade-off space, the ConfigManager can optimize the third (e.g. minimize max read latency across front-ends given write latency and storage cost SLOs). Given the stability of latencies observed between data centers in the cloud both in prior work [34] and in our measurements,[2] and since our current implementation assumes an object's access set is unchanged after it is created, we defer reconfiguration of an object's data sites [19] to future work.

[2]In six months of latency measurements between all pairs of Azure data centers, we observe less than 6% change in median latency from month to month for any data center pair and less than 10% difference between 90th percentile and median latency within each month for most pairs.

Executing reads and writes. Unlike typical applications of Paxos, our use of erasure coding prevents servers from processing the contents of Paxos logs. Instead of separating application and Paxos state, we maintain one Paxos log for every key and aggressively prune old log entries. In order to execute a write request, a Proxy VM initiates Phase 1 of Paxos and waits for the delegate to run Phase 2. If the operation times out, the Proxy VM assumes the delegate has failed and executes both phases itself. Once Phase 2 successfully completes, the Proxy VM notifies the client and asynchronously informs learners so that they may commit their local state and garbage collect old log entries. The read path is simpler: a Proxy VM fetches the associated Paxos state and reconstructs the latest value before returning to the client. If the latest state happens to be uncommitted, then the Proxy VM issues a write-back to guarantee consistency.

5 Evaluation

We evaluate PANDO in two parts. First, in a measurement-based analysis, we estimate PANDO's benefits over prior solutions for enabling strongly consistent distributed storage. We quantify these benefits not only with respect to latency and cost separately, but also the extent to which PANDO helps bridge the gap to the lower bound in the latency–cost tradeoff space (§2). Second, we deploy our prototype key-value store and compare latency and throughput characteristics under microbenchmarks and an application workload. The primary takeaways from our evaluation are:

- Compared to the union of the best available replication- and erasure coding-based approaches, PANDO reduces the median gap to the lower bound by 88% in the read latency–write latency–storage overhead tradeoff space.
- Compared to EPaxos, given bounds on any two of storage overhead, read latency, and write latency, PANDO can improve read latency by 12–31% and reduce dollar costs (for storage, compute, and data transfers) by 6–46%, while degrading write latency by at most 3%.
- In a geo-distributed deployment on Azure, PANDO offers 18–62% lower read latencies than EPaxos and can reduce 95th percentile latency for two GitLab operations by 19–60% over EPaxos and RS-Paxos.

5.1 Measurement-based analysis

Setup. Our analysis uses network latencies between all pairs of 25 Microsoft Azure data centers. We categorize access sets (the subset of data centers from which an object is accessed) into four types: North America (NA), North America & Europe (NA-EU), North America & Asia (NA-AS), and Global (GL). For NA and NA-EU, we use 200 access sets chosen randomly. For NA-AS and GL, we first filter front-end data centers so that they are at least 20 ms apart, and then sample 200 random access sets. In all cases, we consider all 25 Azure data centers as potential data sites.

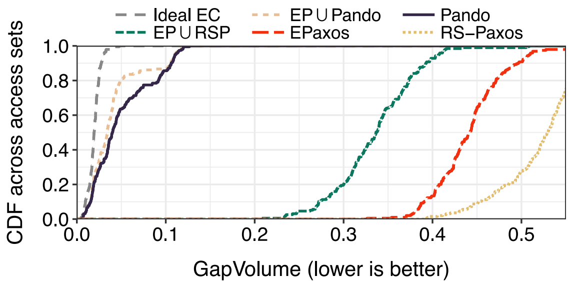


Figure 7: **For NA-AS access sets, comparison of GapVolume with PANDO to EPaxos and RS-Paxos individually and their union (EP ∪ RSP). In addition, we evaluate EP ∪ PANDO (the union of EPaxos and PANDO) and Ideal EC (a hypothetical Paxos variant that supports erasure coding, one-round writes, and 1-split intersection across quorums).**

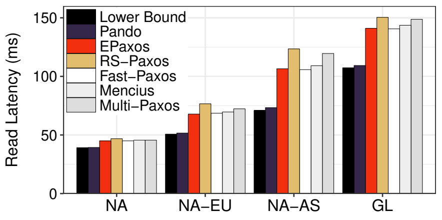
We compare PANDO to four replication-based approaches (EPaxos [51], Fast Paxos [42], Mencius [49], and Multi-Paxos [40]) and the only prior approach which can enable conditional updates on erasure-coded data (RS-Paxos [53]). We refer to the union of EPaxos and RS-Paxos (i.e., use either approach to satisfy the desired SLOs) as EP ∪ RSP.

Metrics. Our analysis looks at three types of metrics: 1) read and write latency (in either case, we estimate the max latency seen by any front-end in the access set) and storage overhead (size of the data stored divided by size of user data); 2) *GapVolume*, a metric which captures the gap in the three-dimensional read latency–write latency–storage overhead tradeoff space between the lower bound (described in §2) and the approach in question; and 3) total dollar cost as the sum of compute, storage, data transfer, and operation costs necessary to support reads and writes.

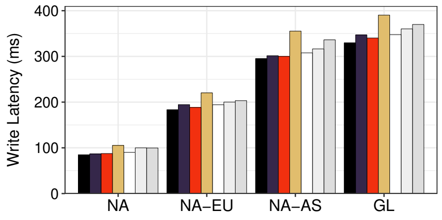
5.1.1 Impact on Achievable Tradeoffs

We use GapVolume to evaluate how close each approach is to the lower bound (§2.3). For any access set, we compute GapVolume with a specific consensus approach as the gap in the (read, write, storage) tradeoff space between the surfaces represented by the lower bound and by tradeoffs achievable with this consensus approach. We normalize this gap relative to the volume of the entire theoretically feasible tradeoff space, i.e., the portion of the tradeoff space above the lower bound surface. For every access set, we cap read and write latencies at values that are achievable with all approaches, and we limit storage overhead to a maximum of 7 as higher values are unlikely to be tenable in practice.

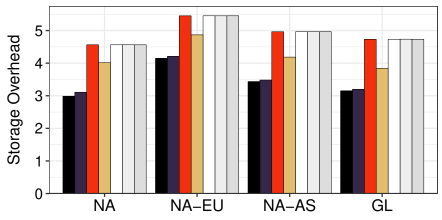
Proximity to lower bound. Figure 7 shows that PANDO significantly reduces GapVolume compared to EPaxos and RS-Paxos for access sets of type NA-AS. We do not show results for other replication-based approaches because they are subsumed by EPaxos, i.e., every combination of SLOs that is achievable with Mencius, Fast Paxos, and Multi-Paxos is also achievable with EPaxos. PANDO lowers median GapVolume to 4%, compared to 53% with RS-Paxos and 44% with EPaxos. Even with EP ∪ RSP (i.e., use two



(a) Read Latency



(b) Write Latency



(c) Storage Overhead

Figure 9: **Average performance across different metrics. Lower is better in all plots. For each metric, we pick SLO combinations for the other two metrics that are achievable with all approaches. For each such SLO pair, we estimate the minimum value of the metric achievable with each approach. We then take the geometric mean across all access sets and SLO pairs.**

GapVolume	NA	NA-EU	NA-AS	GL
PANDO	0.06	0.07	0.04	0.07
EP ∪ RSP	0.37	0.40	0.34	0.34
EPaxos	0.44	0.48	0.44	0.49
RS-Paxos	0.52	0.59	0.53	0.48

Table 1: **GapVolume for median access set of various types. Lower values are better; imply closer to the lower bound.**

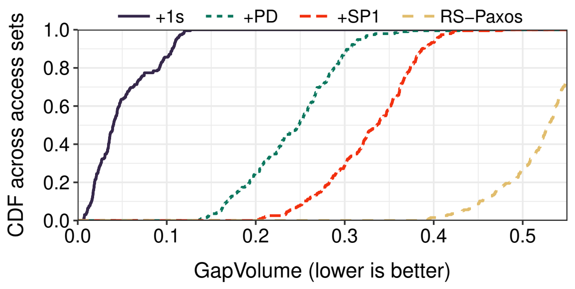


Figure 8: **For access sets of type NA-AS, contributions of each of PANDO's techniques in reducing GapVolume. SP1 = small Phase 1, PD = partial delegation, 1s = 1-split overlap.**

significantly different designs to realize different tradeoffs), median GapVolume remains at 34%. Table 1 shows similar benefits for NA, NA-EU, and GL access sets.

Moreover, EP ∪ PANDO (i.e., SLO combinations achievable with any of EPaxos or PANDO) is only marginally closer to the lower bound (i.e., has lower GapVolume) than PANDO, and that too only for some access sets. The few SLO combinations that EPaxos can achieve but not PANDO all have low write latency SLOs, in which case no choice of delegate can help PANDO overcome the overheads of two-round writes.

Utility of individual techniques. Figure 8 shows that each of the techniques used in PANDO contribute to the GapVolume reductions. For the median access set, using small Phase 1 quorums reduces GapVolume over RS-Paxos by 36%, adding partial delegation reduces GapVolume by a further 16%, and finally incorporating 1-split intersection reduces GapVolume by an additional 39%. When examining the improvements for each access set, we observe that both small Phase 1 quorums and 1-split intersection help across all access sets by reducing quorum size requirements. Similarly, we find that partial delegation typically improves GapVolume, indicating that some data sites are often closer to Phase 1 and Phase 2 quorums than the front-end.

Obstacles to matching the lower bound. From the gap between PANDO and Ideal EC in Figure 7, we surmise that most of the remaining gap between PANDO and the lower bound could be closed if one-round writes on erasure-coded data were feasible. Addressing any potential sub-optimality thereafter likely requires realizing the lower bound's flexibility with regards to varying the fraction of an object's data across sites (e.g., by using a different erasure coding strategy than Reed-Solomon coding) and varying quorum sizes across front-ends.

5.1.2 Latency and Storage Improvements

Figure 9 examines improvements in each of read latency, write latency, and storage overhead independently. To do this for read latency, we first identify all (write, storage) SLO pairs that are achievable by all candidate approaches. For each such pair, we then estimate the lowest read latency bound that is satisfiable with each approach. We take the geometric mean [31] across all feasible (write, storage) SLO pairs for all access sets to compare PANDO's performance relative to other approaches. We perform similar computations for write latency and storage overhead.

We find that PANDO achieves 12–31% lower read latency, 0–3% higher write latency, and 22–32% lower storage overhead than EPaxos across all types of access sets. Although PANDO executes writes in two phases, the use of small Phase 1 quorums plus partial delegation provides similar write latency as EPaxos. In all cases, EPaxos outperforms Fast Paxos, Mencius, and Multi-Paxos. Compared to RS-Paxos, PANDO reduces read latency by 15–40%, write latency by 11–17%, and storage overhead by 13–22%.

Latency under failures. Figure 10 compares the read latency bounds satisfiable with PANDO and EPaxos when any one data center is unavailable. During failures, a front-end may need to contact more distant data sites in order to read or write data. In this case, for the median access set, we observe that PANDO supports a read latency bound which is 110 ms lower than EPaxos. Since erasure coding spreads data more widely than replication for the same storage overhead, there are more nearby sites to fall back on when a failure occurs.

However, erasure coding is not universally helpful in failure scenarios. Upon detecting the loss of its write delegate, a PANDO front-end will locally identify a new one that min-

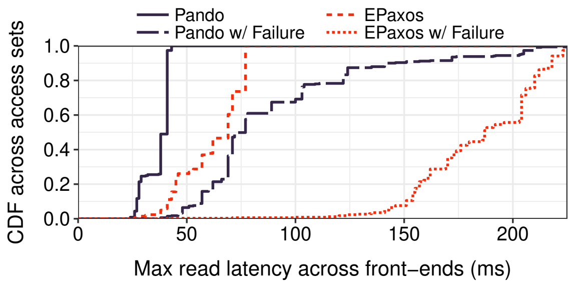


Figure 10: **For access sets of type NA-AS, impact of data center failures on read latency for PANDO and EPaxos (300 ms write SLO, 5× overhead storage SLO).**

imizes latency at the front-end. Still, across NA-AS access sets, median write latency with PANDO is 10% higher than EPaxos when any one data site is unavailable, despite the two approaches having similar latency in the failure-free case. In addition, under permanently data loss, bringing up a replacement data site requires decoding the data of k separate sites instead of fetching the same volume of data from one replica.

5.1.3 Cost

Beyond storage, public cloud providers also charge users for wide-area data transfers, PUT/GET requests to storage, and for virtual machines used to execute RPCs and encode/decode data. These overheads have driven production systems to adopt two key optimizations. First, replication-based systems execute reads by fetching version numbers from remote sites, not data. Second, Paxos-based systems issue writes only to a quorum (or for PANDO, a superset of a write quorum that intersects with all Phase 1a quorums likely to be used). Taking these optimizations into consideration, we now account for these other sources of cost and evaluate PANDO's utility in reducing total cost.

We considered 200 access sets of type NA-AS and set latency SLOs that both RS-Paxos and EPaxos are capable of meeting: 100 ms for read latency and 375 ms for write latency. We derived the CPU cost of Proxy VMs by measuring the throughput achieved in deployments of our prototype system. Using pricing data from Azure CosmosDB [3], we estimated the cost necessary to store 10 TB of data and issue 600M requests, averaged across all access sets; these parameters are based on a popular web service's workload [7] and a poll of typical MySQL deployment sizes [10].

Across several values for mean object size and read-to-write ratio, Figure 11 shows that PANDO reduces overall costs by 6–46% over EPaxos and 35–40% over RS-Paxos. When objects are large, PANDO's cost savings primarily stem from the reduction in the data transferred over the wide-area network. Note that even though EPaxos uses replication, it still requires reading remote data when a copy is not stored at the front-end data center. Whereas, when objects are small, storage fees dominate and PANDO reduces cost primarily due to the lower storage overhead that it imposes. Though erasure coding increases the number of requests

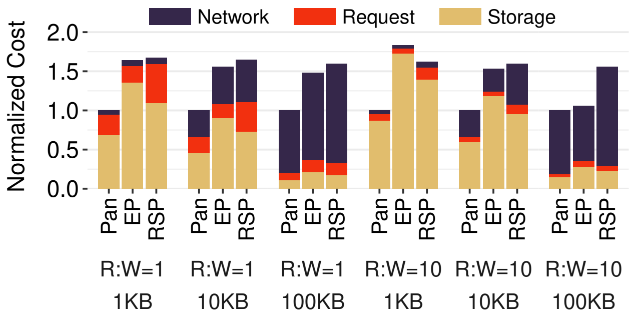


Figure 11: **Comparison of cost for a month in NA-AS to store 10 TB of data and execute 600M requests/month. In all cases, the costs of Proxy VMs (not shown) were negligible, and read and write latency SLOs were set to 100 ms and 375 ms.**

to storage compared to replication, ConfigManager opts to erasure-code data only when the corresponding decrease in storage and data transfer costs help reduce overall cost. Unlike write requests, which have to first write metadata to storage before transferring and writing the data itself, read requests only issue storage operations to fetch data. This leads to greater cost reductions for read-dominated workloads.

5.2 Prototype deployment

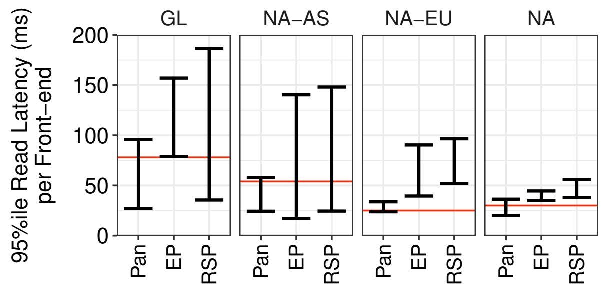
Next, via deployments on Azure, we experimentally compare PANDO versus EPaxos and RS-Paxos. We use our implementations of PANDO and RS-Paxos and the open-source implementation of EPaxos [50]. This experimental comparison helps account for factors missing from our analysis, such as latency variance and contention between requests. We consider one access set of each of our 4 types:

- NA: Central US, East US, North Central US, West US
- NA-EU: Canada East, Central US, North Europe, West Europe
- NA-AS: Central US, Japan West, Korea South
- GL: Australia East, North Europe, SE Asia, West US

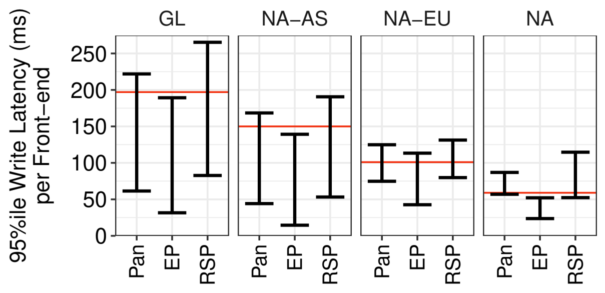
Informed by prior studies of production web service workloads [29, 22], we read and write objects between 1–100 KB in size. Unless stated otherwise, we use A1v2 (1 CPU, 2 GB memory) virtual machines and issue requests using YCSB [28]—a key-value store benchmark.

5.2.1 Microbenchmarks

Tail latency across front-ends. Figure 12 shows 95th percentile read and write latencies for each of the four access sets when running a low contention (zipfian coefficient = 0.1) workload with 1 KB values and a read:write ratio of 1. In all cases, when using PANDO, we observe that the slowest front-end performs similarly to the read latency SLO deemed feasible by ConfigManager. This confirms the low latency variance in the CosmosDB instance at each data center and on the network paths between them. While all approaches achieve sub-55 ms read latency in NA, only PANDO can provide sub-100 ms latency in all regions. In GL, NA-AS, and



(a) Read Latency when Write SLO = 300, 300, 150, and 100 ms for GL, NA-AS, NA-EU, and NA, respectively



(b) Write Latency when Read SLO = 200, 150, 125, and 75 ms for GL, NA-AS, NA-EU, and NA, respectively

Figure 12: **Latency comparison with a low contention workload under a storage SLO of 3× overhead. Red lines represent the lowest latency SLO that ConfigManager identifies as feasible with PANDO. With every approach, in each access set, we measure 95th %ile latency at every front-end and plot the min and max of this value across front-ends. Pan = Pando, EP = EPaxos, and RSP = RS-Paxos.**

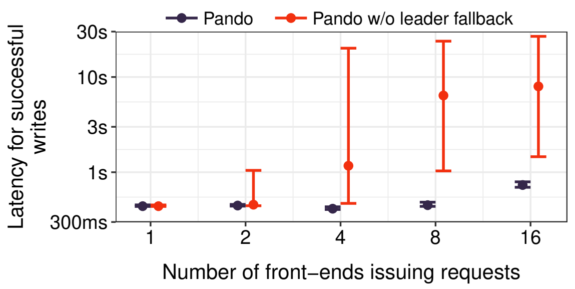


Figure 13: **Write latency comparison under contention using a fully leaderless approach and the leader-based fallback (§3.5). 5th percentile, median, and 95th percentile across 1000 writes are shown. Note logscale on y-axis.**

NA-EU, PANDO improves read latency for the slowest front-end by 39–62% compared to EPaxos. PANDO falls short of the write latency offered by EPaxos but comes close.

Latency under high conflict rates. Although our focus is on workloads with few write conflicts, we seek to bound performance degradation when conflicts occur. To evaluate this, we setup front-ends in 16 Azure data centers spread across five continents. We mimic conflicts by synchronizing a subset of front-ends (assuming low clock skew) to issue writes on the same key and version simultaneously. We show latency for successful conditional writes since other writes will learn the committed value and terminate shortly afterward.

Figure 13 shows that PANDO is effective at bounding latency for writes in the presence of conflicts. Without a leader-based fallback, writes in PANDO may need to be tried many times before succeeding, resulting in unbounded latency growth, e.g., with four concurrent writers, we observe more than 15 proposals for particular (key, version) pairs. In contrast, falling back to a leader ensures that a write succeeds within two write attempts.

Read and write throughput. While erasure coding can decrease bandwidth usage compared to replication, it requires additional computation in the form of coding/decoding and messaging overhead. We quantify the inflection point at which CPU overheads dominate by deploying PANDO in a single data center and measuring the achievable throughput

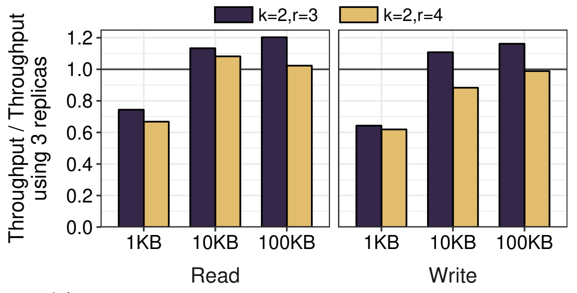


Figure 14: **Per-machine throughput of different erasure coding configurations compared to using 3 replicas.**

with all data in memory. Each server, which stored 1 split or 1 replica, had two Xeon Silver 4114 processors and 192 GB of memory. All servers were connected over a 10 Gbps network with full bisection bandwidth. Across multiple value sizes, we measured the per-server throughput of filling the system with over 20 GB of data and reading it back.

Figure 14 compares the per-machine throughput achieved with 3 replicas to two erasure coding configurations, one with the same storage overhead and another with lower storage overhead. When objects are 10 KB or larger, we find that bandwidth is the primary bottleneck. Because it has identical bandwidth demands as replication, the $(k = 2, r = 4)$ configuration achieves similar read throughput and 0.9–1× the write throughput of replication for objects larger than 10 KB. Whereas, due to its lower bandwidth consumption, the $(k = 2, r = 3)$ configuration offers 1.1-1.2× the throughput of replication for 10 KB–100 KB sized objects. All configurations are CPU-bound with value sizes of 1 KB or smaller. Since replication requires exchanging fewer messages per request than erasure coding, it has lower CPU overhead and can thus achieve higher throughput.

5.2.2 Application Workload

Lastly, we evaluate the utility of PANDO on a geo-distributed deployment of GitLab [6], a software development application that provides source code management, issue tracking, and continuous integration.

Operations and setup. We evaluate the performance of two GitLab operations: listing issues targeting a devel-

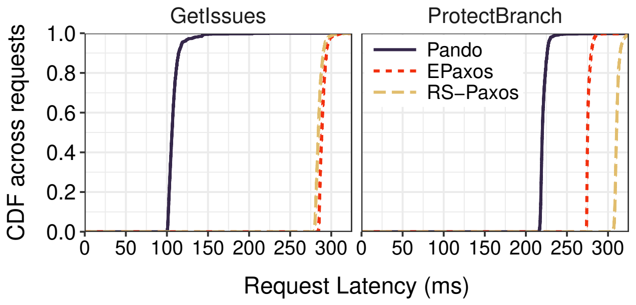


Figure 15: **Latencies for GitLab requests in Central US.**

opment milestone (GetIssues) and (un-)protecting a branch from changes (ProtectBranch). GetIssues fetches a list of issues for the requested milestone and then fetches 20 issues in parallel to display on a page. ProtectBranch reads the current branch metadata then updates its protection status.

We deployed front-ends and storage backends in the NA-AS access set on A2v2 (2 CPU, 4 GB memory) virtual machines, and preloaded the system with 100 projects, each with 20 branches, 10 milestones, and 100 issues. We used a $3\times$ bound on storage overhead and set the write latency SLO to 175 ms. Every front-end executed 1000 GetIssues and ProtectBranch requests in an open loop and selected items using a uniform key distribution.

Performance. Figure 15 shows the latency distribution observed for both operations by the front-end in Central US. PANDO reduces 95th percentile GetIssues latency by over 59% compared to both EPaxos and RS-Paxos. Because ProtectBranch consists of a write in addition to a read operation, it incurs higher latency compared to GetIssues, which consists solely of read operations. Despite this, PANDO is able to lower 95th percentile ProtectBranch latency by 19% over EPaxos and 28% over RS-Paxos.

6 Related work

Geo-distributed storage. While some prior geo-distributed storage systems [46, 45, 47, 66] weaken consistency semantics to minimize latencies and unavailability, PANDO follows others [29, 64, 71, 21] in serving the needs of applications that cannot make do with weak consistency. Compared to efforts focused solely on minimizing latency with any specific replication factor [51, 49, 42], PANDO aims to also minimize the cost necessary to meet latency goals. Unlike systems [13, 70] which focus only on judiciously placing data to minimize cost, we also leverage erasure coding and rethink how to enable consensus on erasure-coded data.

Partial delegation in PANDO is akin to the chaining of RPCs [63] to eliminate wide-area delays. We show that combining this technique with the use of smaller quorums in Phase 1 of Paxos helps a two-round execution approximate the latencies achievable with one-round protocols in a geo-distributed setting.

Erasure-coded storage. Erasure coding has been widely used for protecting data from failures [69], most notably in RAID [58]. While PANDO leverages Reed Solomon codes [60] for storage across data centers, other codes have been used to correct errors in DRAM [33], transmit data over networks [62], and efficiently reconstruct data in cloud storage [67, 38, 61]. In contrast to the typical use of erasure coding for immutable and/or cold data [54, 32, 59, 55, 27], PANDO supports the storage of hot, mutable objects.

Previous protocols [15, 24] that support strong consistency with erasure-coded data provide only atomic register semantics or require two rounds of communication [53]. We show how to enable consensus on geo-distributed erasure-coded data without sacrificing latency. Some systems [26, 27] support strong consistency by erasure coding data but replicating metadata. We chose to not pursue this route to avoid the complexity of keeping the two in sync, as well as to minimize latency and metadata overhead.

Paxos variants. Many variants of Paxos [40] have been proposed over the years [53, 37, 43, 41], including several [51, 42, 49] which enable low latency geo-distributed storage. Compared to Paxos variants that reduce the number of wide-area round trips [51, 49, 42], PANDO lowers latency by reducing the magnitude of delay in each round trip.

Flexible Paxos [37] was the first to observe that Paxos only requires overlap between every Phase 1–Phase 2 quorum pair, and others [16, 56] have leveraged this observation since. All of these approaches make *Phase 2 quorums smaller*, so as to improve throughput and common case latency in settings with high spatial locality. In PANDO, we instead *reduce the size of Phase 1 quorums and reuse these quorums for reads*, thereby enabling previously unachievable tradeoffs between read and write latency bounds in a workload-agnostic manner.

Compression. Data compression is often used to lower the cost of storing data [36, 65, 25] or transferring it over a network [30]. In contrast to erasure coding, the effectiveness of compression depends on both the choice of compression algorithm used as well as the input data [17]. Compression and erasure coding are complementary as data can be compressed and then erasure-coded or vice-versa.

7 Conclusion

Today, geo-distributed storage systems take for granted that data must be replicated across data centers. In this paper, we showed that it is possible to leverage erasure coding to significantly reduce costs while successfully mitigating the associated overheads in wide-area latency incurred for preserving consistency. The key is to rethink how consensus is achieved across the wide-area. Importantly, we showed that the latency versus cost tradeoffs achievable with our approach for enabling consensus, PANDO, are close to optimal.

Acknowledgments. This work was supported in part by the NSF under grants CNS-1563095, CNS-1563849, CNS-1617773, and CNS-1900665.

References

[1] 100% uptime anybody? `http://www.riskythinking.com/articles/article8.php`.

[2] And the cloud provider with the best uptime in 2015 is ... `http://www.networkworld.com/article/3020235/cloud-computing/and-the-cloud-provider-with-the-best-uptime-in-2015-is.html`.

[3] Azure Cosmos DB - globally distributed database service — Microsoft Azure. `https://azure.microsoft.com/en-us/services/cosmos-db/`.

[4] Google Docs. `https://docs.google.com`.

[5] Latency is everywhere and it costs you sales - how to crush it. `http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it`.

[6] The only single product for the complete devops life-cycle - GitLab. `https://about.gitlab.com/`.

[7] Quizlet.com audience insights - Quantcast. `https://www.quantcast.com/quizlet.com#trafficCard`.

[8] ShareLaTeX. `https://sharelatex.com`.

[9] Storage - Intel ISA-L — Intel software. `https://software.intel.com/en-us/storage/ISA-L`.

[10] What is the largest amount of data do you store in MySQL? - Percona database performance blog. `https://www.percona.com/blog/2012/11/09/what-is-the-largest-amount-of-data-do-you-store-in-mysql/`.

[11] Which cloud providers had the best uptime last year? `http://www.networkworld.com/article/2866950/cloud-computing/which-cloud-providers-had-the-best-uptime-last-year.html`.

[12] SLA for Azure Cosmos DB. `https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_3/`, 2019.

[13] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010.

[14] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *International Symposium on Distributed Computing*, pages 108–122. Springer, 2001.

[15] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.

[16] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar. Multileader WAN paxos: Ruling the archipelago with fast consensus. *CoRR*, 2017.

[17] J. Alakuijala, E. Kliuchnikov, Z. Szabadka, and L. Vandevenne. Comparison of brotli, deflate, zopfli, lzma, lzham and bzip2 compression algorithms. *Google Inc.*, 2015.

[18] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *ICSE*, 1976.

[19] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *OSDI*, 2014.

[20] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[21] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[22] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*, 2013.

[23] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[24] V. R. Cadambe, N. Lynch, M. Médard, and P. Musial. A coded shared atomic memory algorithm for message passing architectures. *Distributed Computing*, 30(1):49–73, 2017.

[25] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[26] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)*, 13(3):25, 2017.

[27] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure coding objects across global data centers. In *USENIX ATC*, 2017.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[29] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[30] R. Fielding and J. Reschke. RFC 7230: Hypertext transfer protocol (HTTP/1.1): Message syntax and routing. *Internet Engineering Task Force (IETF)*, 2014.

[31] P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *CACM*, 1986.

[32] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.

[33] R. W. Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[34] O. Haq, M. Raja, and F. R. Dogar. Measuring and improving the reliability of wide-area cloud paths. In *WWW*, 2017.

[35] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), 1991.

[36] D. R. Horn, K. Elkabany, C. Lesniewski-Lass, and K. Winstein. The design, implementation, and deployment of a system to transparently compress hundreds of petabytes of image files for a file-storage service. In *NSDI*, 2017.

[37] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. In *OPODIS*, 2016.

[38] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In *USENIX ATC*, 2012.

[39] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[40] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[41] L. Lamport. Generalized consensus and paxos. 2005.

[42] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[43] L. Lamport and M. Massa. Cheap paxos. In *DSN*, 2004.

[44] B. Lampson. The abcd's of paxos. In *PODC*, 2001.

[45] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[46] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[47] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[48] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: Measuring and understanding consistency at Facebook. In *SOSP*, 2015.

[49] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI*, 2008.

[50] I. Moraru. Epaxos. `https://github.com/efficient/epaxos`, 2014. commit 791b115.

[51] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *SOSP*, 2013.

[52] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *SoCC*, 2014.

[53] S. Mu, K. Chen, Y. Wu, and W. Zheng. When Paxos meets erasure code: Reduce network and storage cost in state machine replication. In *HPDC*, 2014.

[54] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's warm BLOB storage system. In *OSDI*, 2014.

[55] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. OceanStore: An architecture for global-scale persistent storage. In *OSDI*, 2014.

[56] F. Nawab, D. Agrawal, and A. El Abbadi. DPaxos: Managing data closer to users for low-latency and mobile applications. In *SIGMOD*, 2018.

[57] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[58] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.

[59] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *OSDI*, 2016.

[60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[61] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. 2013.

[62] A. Shokrollahi. Raptor codes. *IEEE/ACM Transactions on Networking (TON)*, 14(SI):2551–2567, 2006.

[63] Y. J. Song, M. K. Aguilera, R. Kotla, and D. Malkhi. RPC chains: Efficient client-server communication in geodistributed systems. In *NSDI*, 2009.

[64] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[65] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, 2005.

[66] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[67] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, S. Hussain, and S. Nandi. Clay codes: Moulding MDS codes to yield an MSR code. In

FAST, 2018.

[68] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[69] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS*, 2002.

[70] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[71] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, 2013.

A The PANDO write protocol: specification and proof of correctness

In this section, we focus on how PANDO achieves consensus on a *single value* and prove that it matches the guarantees provided by Paxos. Other functionality used in our paper is layered on top of this base as follows:

- **Mutating values.** As with Multi-Paxos, we build a distributed log of values and run PANDO on each entry of the log. We only ever attempt a write for version i if we know that $i-1$ has already been chosen. This invariant ensures that the log is contiguous, and that all but possibly the latest version have been decided.
- **Partial delegation of writes.** One of the key optimizations used in PANDO is to execute Phase 1 and Phase 2 on different nodes (§3.3.2). We achieve this without sacrificing fault tolerance as follows. Each proposer is assigned an id (used for Lamport clocks), but we additionally assign a proposer id to each (proposer, delegate) pair. When executing a write using partial delegation, we simply direct responses accordingly, and have the proposer inform the delegate about which value to propose (unless one was recovered, in which case the delegate has to inform the proposer about the change). In case the delegate fails, a proposer can always choose to execute a write operation normally, and because it uses a different proposer id in this case, it will look as though the write from the proposer and the write from the (proposer, delegate) pair are writes from two separate nodes. We already prove (§A.1) that PANDO maintains consistency in this case.
- **One round reads.** As with other consensus protocols, we support (common-case) one-round reads by adding a third, asynchronous phase to writes that broadcasts which value was chosen and caches this information at each acceptor. Upon executing a read at a Phase 1a quorum, we check to see if any acceptor knows whether a value has already been chosen. If we find such a value, we try to reconstruct it and fall back on the larger Phase 1b quorum in case there are not enough splits present in the Phase 1a quorum. Otherwise, we follow the write path, but propose a value only if we were able to recover one (else none have been chosen). We maintain linearizability with this approach because the task of resolving uncertainty is done via the write path.
- **Fallback to leader.** In PANDO, front-ends directly execute writes unless a conflict is observed, in which case they defer the request to a leader (§3.5). From the perspective of the consensus protocol, the leader is just another proposer, so no consistency issues may arise even if multiple leaders exist. However, PANDO prevents non-leader front-ends from attempting writes more than once which can lead to unavailability if the leader fails. It is up to the leader election mechanism to quickly elect a new leader when the the current one fails.

$A.ppn$	Promised proposal no. stored at acceptor A
$A.apn$	Accepted proposal no. stored at acceptor A
$A.vid$	Accepted value id stored at acceptor A
$A.vlen$	Accepted value length stored at acceptor A
$A.vsplit$	Accepted value split stored at acceptor A
vid_v	Unique id for v, typically a hash or random number
$vlen_v$	Length of v (to remove padding)
$\text{Split}(v, A)$	(Computed on proposers) The erasure-coded split associated with acceptor A

Figure 16: Summary of notation.

PANDO's consistency and liveness properties rely on certain quorum constraints being met. We describe the constraints below under the assumption that data is partitioned into k splits (Constraint 3 needed only if Phase 1a quorums are used for reads):

1. The intersection of any Phase 1a and Phase 2 quorums contains at least 1 split.
2. The intersection of any Phase 1b and Phase 2 quorums contains at least k splits.
3. A Phase 1a quorum must contain at least k splits.
4. After f nodes fail, at least one Phase 1b and Phase 2 quorum must consist of nodes that are available.

Below is pseudocode for the PANDO write protocol.

Phase 1 (Prepare-Promise)

```
Proposer $P$ initiates a write for value $v$:
  1. Select a unique proposal number $p$ (typically done using Lamport clocks).
  2. Broadcast $\text{Prepare}(p)$ messages to all acceptors.

Acceptor $A$, upon receiving $\text{Prepare}(p)$ message from Proposer $P$:
  3. If $p > A.ppn$ then set $A.ppn \leftarrow p$ and reply $\text{Promise}(A.apn, A.vid, A.vlen, A.vsplit)$.
  4. Else reply NACK.

Proposer $P$, upon receiving Promise messages from a Phase 1a quorum:
  5. If the values in all Promise responses are NULL, then skip to Phase 2 with $v' \leftarrow v$.

Proposer $P$, upon receiving Promise messages from a Phase 1b quorum:
```

6. Iterate over all Promise responses sorted in decreasing order of their apn.
   (a) If there are at least k splits for value w associated with apn, recover the value w (using the associated $vlen$ and $vsplit$s) and continue to Phase 2 with $v' \leftarrow w$.
7. If no value was recovered, continue to Phase 2 with $v' \leftarrow v$.

<u>*Phase 2 (Propose-Accept)*</u>

Proposer P, initiating Phase 2 to write value v' with proposal number p:

8. If no value was recovered in Phase 1, set $vid_{v'} = hash(v)$ (or some other unique number, see Figure 16). If a value was recovered, use the existing $vid_{v'}$.
9. Broadcast $\text{Propose}(p, vid_{v'}, vlen_{v'}, \text{Split}(v', A))$ messages to all acceptors.

Acceptor A, upon receiving $\text{Propose}(p, vid, vlen, vsplit)$ **from a Proposer P:**

10. If $p < A.ppn$ reply NACK
11. $A.ppn \leftarrow p$
12. $A.apn \leftarrow p$
13. $A.vid \leftarrow vid$
14. $A.vlen \leftarrow vlen$
15. $A.vsplit \leftarrow vsplit$
16. Reply $\text{Accept}(p)$

Proposer P, upon receiving $\text{Accept}(p)$ **messages from a Phase 2 quorum:**

17. P now knows that v' was chosen, and can check whether the chosen value v' differs from the initial value v or not.

A.1 Proof of correctness

Definitions. We let $\mathcal{A}$ refer to the set of all acceptors and use $\mathcal{Q}_a$, $\mathcal{Q}_b$, and $\mathcal{Q}_2$ refer to the sets of Phase 1a, Phase 1b, and Phase 2 quorums, respectively. Using this notation, we restate our quorum assumptions:

$$Q \subseteq \mathcal{A} \qquad \forall Q \in \mathcal{Q}_a \cup \mathcal{Q}_b \cup \mathcal{Q}_2 \qquad (1)$$
$$|Q_a \cap Q_2| \geq 1 \qquad \forall Q_a \in \mathcal{Q}_a, Q_2 \in \mathcal{Q}_2 \qquad (2)$$
$$|Q_b \cap Q_2| \geq k \qquad \forall Q_b \in \mathcal{Q}_b, Q_2 \in \mathcal{Q}_2 \qquad (3)$$

Definition 1. *A value is* ***chosen*** *if there exists a Phase 2 quorum of acceptors that all agree on the identity of the value and store splits corresponding to that value.*

We now show that the PANDO write protocol provides the same guarantees as Paxos:

- **Nontriviality.** Any chosen value must have been proposed by a proposer.
- **Liveness.** A value will eventually be chosen provided that RPCs complete before timing out and all acceptors in at least one Phase 1b and Phase 2 quorum are available.
- **Consistency.** At most one value can be chosen.
- **Stability.** Once a value is chosen, no other value may be chosen.

Theorem 1. *(Nontriviality)* PANDO *will only choose values that have been proposed.*

Proof. By definition, a value can only be chosen if it is present at a Phase 2 quorum of acceptors. Values are only stored at acceptors in response to Propose messages initiated by proposers. □

Theorem 2. *(Liveness)* PANDO *will choose a a value provided that RPCs complete before timing out and all acceptors in at least one Phase 1b and Phase 2 quorum are available.*

Proof. Let t refer to the (maximum) network and execution latency for an RPC. Since PANDO has two rounds of execution, a write can complete within $2t$ as long as a requested is uncontended. If all proposers retry RPCs using randomized exponential backoff, a time window of length $\geq 2t$ will eventually open where only a Proposer P is executing. Since no other proposer is sending any RPCs during this time, both Phase 1 and Phase 2 will succeed for Proposer P. □

Following the precedence of [37], we will show that PANDO provides both consistency and stability by proving that it provides a stronger guarantee.

Lemma 1. *If a value v is chosen with proposal number p, then for any proposal with proposal number $p' > p$ and value v', $v' = v$.*

Proof. Recall that PANDO proposers use globally unique proposal numbers (Line 1); this makes it impossible for two different proposals to share a proposal number p. Therefore, if two proposals are both chosen, they must have different proposal numbers. If $v' = v$ then we trivially have the desired property. Therefore, assume $v' \neq v$.

Without loss of generality, we will consider the smallest p' such that $p' > p$ and $v' \neq v$ (*minimality assumption*). We will show that this case always results in a contradiction: either the Prepare messages for p' will fail (and thus no Propose messages will ever be sent) or the proposer will adopt and re-propose value v.

Let $Q_{2,p}$ be the Phase 2 quorum used for proposal number p, and $Q_{a,p'}$ be the Phase 1a quorum used for p'. By Quorum Property 2, we know that $|Q_{2,p} \cap Q_{a,p'}|$ is non-empty. We will now look at the possible ordering of events at each acceptor A in the intersection of these two quorums ($Q_{2,p}$ and $Q_{a,p'}$):

- Case 1: A receives $\text{Prepare}(p')$ before $\text{Propose}(p, \ldots)$.

  The highest proposal number at A would be $p' > p$ by the time $\text{Propose}(p, \ldots)$ was processed, and so A would reject $\text{Propose}(p, \ldots)$. However, we know that this is not the case since $A \in Q_{2,p}$, so this is a contradiction.

- Case 2: A receives $\text{Propose}(p, \ldots)$ before $\text{Prepare}(p')$.

  The last promised proposal number at A is q such that $p \leq q < p'$ ($q > p'$ would be a contradiction since $\text{Prepare}(p')$ would fail even though $A \in Q_{a,p'}$). By our minimality assumption, we know that all proposals z such that $p \leq z < p'$ fail or re-propose v. Therefore, the acceptor A responds with $\text{Promise}(q, vid_v, \ldots)$.

At this point, the proposer has received at least one Promise message with a non-empty value. Therefore, it does not take the Phase 1 fast path and waits until it has heard from a Phase 1b quorum (denoted $Q_{b,p'}$). Using the same logic as above, the proposer for p' will receive a minimum of k Promise messages each referencing value v since there are k acceptors in $Q_{b,p'} \cap Q_{2,p}$ (Quorum Property 3). Since the proposer has a minimum of k responses for v, it can reconstruct value v. Let q denote the highest proposal number among all k responses.

Besides those in $Q_{b,p'} \cap Q_{2,p}$, other acceptors in $Q_{b,p'}$ may return values that differ from v. We consider the proposal number q' for each of these accepted values:

- Case 1: $q' < q$. The proposer for p' will ignore the value for q' since it uses the highest proposal number for which it has k splits.

- Case 2: $p' < q'$. Not possible since $\text{Prepare}(p')$ would have failed.

- Case 3: $p < q' < p'$. This implies that a $\text{Propose}(q', v'')$ was issued where $v'' \neq v$. This violates our minimality assumption.

Therefore, the proposer will adopt value v since it can reconstruct it (the proposer has k splits from the acceptors in $Q_{b,p'} \cap Q_{2,p}$ alone) and the highest returned proposal number references it. This contradicts our assumption that $v' \neq v$. □

Theorem 3. *(Consistency)* PANDO *will choose at most one value.*

Proof. Assume that two different proposals with proposal numbers p and q are chosen. Since proposers use globally unique proposal numbers, $p \neq q$. This implies that one of the proposal numbers is greater than the other, assume that $q > p$. By Lemma 1, the two proposals write the same value. □

Theorem 4. *(Stability) Once a value is chosen by* PANDO, *no other value may be chosen.*

Proof. The proposal numbers used for any two chosen proposals will not be equal. Thus, with the additional assumption that acceptors store their state in durable storage, this follows immediately from Lemma 1. □

B TLA+ specification for Pando reads and writes

In addition to our proof of correctness for Pando's write path, we have model checked Pando's correctness using TLA+ [39]. The purpose of this exercise was to *mechanically verify* Pando's safety guarantees under a number of scenarios.

We checked the following invariants: consistency and stability for writes, that any value marked chosen at an acceptor was indeed chosen, and that successful reads only ever returned chosen values. The configurations modeled used 2–3 proposers (and readers) that could write (read) 2–3 values to (from) 4–6 acceptors when splitting the data into 2–4 splits. We set up 2–3 quorums of each type (Phase 1a, Phase 1b, and Phase 2).

The TLA+ model checker considers all possible histories including those with message reordering and arbitrary (or infinite) delay in delivering messages. When run on the specification for Pando (below) and the configurations listed earlier, no invariant violations were found.

```
----------------------------- MODULE Pando -----------------------------
EXTENDS Integers, TLC, FiniteSets

CONSTANTS Acceptors, Ballots, Values,
          Quorum1a, Quorum1b, Quorum2, K

ASSUME QuorumAssumption ≜
          ∧ Quorum1a ⊆ SUBSET Acceptors
          ∧ Quorum1b ⊆ SUBSET Acceptors
          ∧ Quorum2  ⊆ SUBSET Acceptors
          Overlap of 1
          ∧ ∀ QA ∈ Quorum1a :
              ∀ Q2 ∈ Quorum2 :
                Cardinality(QA ∩ Q2) ≥ 1
          Overlap of K
          ∧ ∀ QB ∈ Quorum1b :
              ∀ Q2 ∈ Quorum2 :
                Cardinality(QB ∩ Q2) ≥ K

VARIABLES msgs,       The set of messages that have been sent
          maxPBal,    maxPBal[a] is the highest promised ballot (proposal number) at acceptor a
          maxABal,    maxABal[a] is the highest accepted ballot (proposal number) at acceptor a
          maxVal,     maxVal[a] is the value for maxABal[a] at acceptor a
          chosen,     chosen[a] is the value that acceptor a heard was chosen (or else is None)
          readLog     readLog[b] is the value that was read during ballot b

vars  ≜ ⟨msgs, maxPBal, maxABal, maxVal, chosen, readLog⟩
None  ≜ CHOOSE v : v ∉ Values

Type invariants.
Messages ≜
          [type : {"prepare"}, bal : Ballots]
    ∪     [type : {"promise"}, bal : Ballots, maxABal : Ballots ∪ {−1},
           maxVal : Values ∪ {None}, acc : Acceptors,
           chosen  : Values ∪ {None}]
    ∪     [type : {"propose"}, bal : Ballots, val : Values ∪ {None},
           op : {"R", "W"}]
    ∪     [type : {"accept"}, bal : Ballots, val : Values, acc : Acceptors,
           op : {"R", "W"}]
    ∪     [type : {"learn"}, bal : Ballots, val : Values]

TypeOK ≜ ∧ msgs     ∈ SUBSET Messages
         ∧ maxABal  ∈ [Acceptors → Ballots ∪ {−1}]
         ∧ maxPBal  ∈ [Acceptors → Ballots ∪ {−1}]
         ∧ maxVal   ∈ [Acceptors → Values ∪ {None}]
```

```
        $\land\ chosen \quad \in [Acceptors \rightarrow Values \cup \{None\}]$
        $\land\ readLog \quad \in [Ballots \rightarrow Values \cup \{None\}]$
        $\land\ \forall\, a \quad \in Acceptors : maxPBal[a] \geq maxABal[a]$
```

Initial state.

```
$Init \triangleq$ $\land\ msgs \quad = \{\}$
       $\land\ maxPBal = [a \in Acceptors \mapsto -1]$
       $\land\ maxABal = [a \in Acceptors \mapsto -1]$
       $\land\ maxVal \quad = [a \in Acceptors \mapsto None]$
       $\land\ chosen \quad = [a \in Acceptors \mapsto None]$
       $\land\ readLog \quad = [b \in Ballots \mapsto None]$
```

Send message m.

```
$Send(m) \triangleq msgs' = msgs \cup \{m\}$
```

Prepare: The proposer chooses a ballot id and broadcasts prepare requests to all acceptors.
All writes start here.

```
$Prepare(b) \triangleq$ $\land\ \neg\exists\, m \in msgs : (m.type = \text{“prepare”}) \land (m.bal = b)$
                 $\land\ Send([type \mapsto \text{“prepare”},\ bal \mapsto b])$
                 $\land\ \text{UNCHANGED}\ \langle maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog\rangle$
```

Promise: If an acceptor receives a prepare request with ballot id greater than that of any prepare request which it has already responded to, then it responds to the request with a promise. The promise reply contains the proposal (if any) with the highest ballot id that it has accepted.

```
$Promise(a) \triangleq$
  $\exists\, m \in msgs :$
    $\land\ m.type = \text{“prepare”}$
    $\land\ m.bal > maxPBal[a]$
    $\land\ Send([type \mapsto \text{“promise”},\ acc \mapsto a,\ bal \mapsto m.bal,$
             $maxABal \mapsto maxABal[a],\ maxVal \mapsto maxVal[a],$
             $chosen \mapsto chosen[a]])$
    $\land\ maxPBal' = [maxPBal\ \text{EXCEPT}\ ![a] = m.bal]$
    $\land\ \text{UNCHANGED}\ \langle maxABal,\ maxVal,\ chosen,\ readLog\rangle$
```

Propose (fast path): The proposer waits until it collects promises from a Phase 1a quorum of acceptors. If no previous value is found, then the proposer can skip to Phase 2 with its own value.

```
$ProposeA(b) \triangleq$
  $\land\ \neg\exists\, m \in msgs : (m.type = \text{“propose”}) \land (m.bal = b)$
  $\land\ \exists\, v \in Values\ :$
    $\land\ \exists\, Q \in Quorum1a :$
      $\text{LET}\ Q1Msgs \triangleq \{m \in msgs :\ \land\ m.type = \text{“promise”}$
                                          $\land\ m.bal \quad = b$
                                          $\land\ m.acc \quad \in Q\}$
      $\text{IN}$
        Check for promises from all acceptors in Q
        $\land\ \forall\, a \in Q : \exists\, m \in Q1Msgs : m.acc = a$
        Make sure no previous vals have been returned in promises
        $\land\ \forall\, m \in Q1Msgs : m.maxABal = -1$
    $\land\ Send([type \mapsto \text{“propose”},\ bal \mapsto b,\ val \mapsto v,\ op \mapsto \text{“W”}])$
  $\land\ \text{UNCHANGED}\ \langle maxPBal,\ maxABal,\ maxVal,\ chosen,\ readLog\rangle$
```

Propose (slow path): The proposer waits for promises from a Phase 1b quorum of acceptors. If no value is found accepted, then the proposer can pick its own value for the next phase. If any accepted coded split is found in one of the promises, the proposer detects whether there are at least K splits (for the particular value) in these promises. Next, the proposer picks up the recoverable value with the highest ballot, and uses it for next phase.

```
$ProposeB(b) \triangleq$
  $\land\ \neg\exists\, m \in msgs : (m.type = \text{“propose”}) \land (m.bal = b)$
  $\land\ \exists\, Q \in Quorum1b :$
    $\text{LET}\ Q1Msgs \triangleq \{m \in msgs :\ \land\ m.type = \text{“promise”}$
```

```
                                    $\land\ m.bal\ \ = b$
                                    $\land\ m.acc\ \ \in Q\}$
        $Q1Vals\ \ \triangleq\ [v\ \ \in Values \cup \{None\} \mapsto$
                        $\{m \in Q1Msgs : m.maxVal = v\}]$
    IN
      Check that all acceptors from $Q$ responded
      $\land\ \forall a \in Q : \exists m \in Q1Msgs : m.acc = a$
      $\land\ \exists v \in Values :$
          $\land$  No recoverable value, use anything
            $\lor\ \forall vv \in Values : Cardinality(Q1Vals[vv]) < K$
            Check if $v$ is recoverable and of highest ballot
            $\lor$  Use previous value if $K$ splits exist
               $\land\ Cardinality(Q1Vals[v]) \geq K$
               $\land\ \exists m \in Q1Vals[v] :$
                    Ensure no other recoverable value has a higher ballot
                    $\land\ \forall mm \in Q1Msgs :$
                         $\lor\ m.bal \geq mm.bal$
                         $\lor\ Cardinality(Q1Vals[mm.maxVal]) < K$
          $\land\ Send([type \mapsto$ "propose", $bal \mapsto b,\ val \mapsto v,\ op \mapsto$ "W"$])$
  $\land$ UNCHANGED $\langle maxPBal, maxABal, maxVal, chosen, readLog\rangle$
```

Phase 2: If an acceptor receives an accept request with ballot i, it accepts the proposal unless it has already responded to a prepare request having a ballot greater than it does.

```
$Accept(a)\ \triangleq$
  $\land\ \exists m \in msgs :$
    $\land\ m.type =$ "propose"
    $\land\ m.bal \geq maxPBal[a]$
    $\land\ maxABal' = [maxABal$ EXCEPT $![a] = m.bal]$
    $\land\ maxPBal' = [maxPBal$ EXCEPT $![a] = m.bal]$
    $\land\ maxVal'\ \ = [maxVal\ $ EXCEPT $![a]\ = m.val]$
    $\land\ Send([type \mapsto$ "accept", $bal \mapsto m.bal,\ acc \mapsto a,\ val \mapsto m.val,$
              $op \mapsto m.op])$
  $\land$ UNCHANGED $\langle chosen, readLog\rangle$
```

ProposerEnd: If the proposer receives acknowledgements from a Phase 2 quorum, then it knows that the value was chosen and broadcasts this.

```
$ProposerEnd(b)\ \triangleq$
  $\land\ \exists v \in Values :$
    $\land\ \exists Q \in Quorum2 :$
      LET $Q2msgs\ \triangleq\ \{m \in msgs :\ \land\ m.type =$ "accept"
                                    $\land\ m.bal\ \ = b$
                                    $\land\ m.val\ \ = v$
                                    $\land\ m.acc \in Q\}$
      IN
        Check for accept messages from all members of $Q$
        $\land\ \forall a \in Q : \exists m \in Q2msgs : m.acc = a$
        If this was in response to a read, log the result
        $\land$  Read: log the result
          $\lor\ \land\ \exists m \in Q2msgs : m.op =$ "R"
             $\land\ readLog' = [readLog$ EXCEPT $![b] = v]$
          Write: don't log the result
          $\lor\ (\forall m \in Q2msgs : m.op =$ "W" $\land$ UNCHANGED $\langle readLog\rangle)$
    $\land\ Send([type \mapsto$ "learn", $bal \mapsto b,\ val \mapsto v])$
  $\land$ UNCHANGED $\langle maxABal, maxPBal, maxVal, chosen\rangle$
```

Learn: A proposer has announced that value v is chosen.

```
$Learn(a) \triangleq$
  $\land\ \exists\, m \in msgs :$
    $\land\ m.type =$ "learn"
    Process accept before learn, needed for ReadInv, not the protocol
    $\land\ maxABal[a] \geq m.bal$
    $\land\ chosen' = [chosen$ EXCEPT $![a] = m.val]$
  $\land$ UNCHANGED $\langle msgs, maxPBal, maxABal, maxVal, readLog \rangle$
```

Count how many splits of v we have received.

```
$CountSplitsOf(resps, v) \triangleq Cardinality(\{m \in resps : m.maxVal = v\})$
```

FastRead: Check if any value returned from a Phase 1a quorum was chosen. If we have enough splits to reconstruct that value, then return immediately. If not, wait for Phase 1b quorum. If we have a value that was marked chosen, return. Otherwise, perform a write-back.

```
$FastRead(b) \triangleq$
  $\land\ \neg\exists\, m \in msgs : (m.type =$ "propose"$) \land (m.bal = b)$
  $\land$
    Fastest path: Phase 1a quorum has $k$ splits and the value is chosen
    $\lor\ \land\ \exists\, Q \in Quorum1a :$
          LET $RMsgs \triangleq \{m \in msgs :\ \land\ m.type =$ "promise"
                                        $\land\ m.bal\ \ = b$
                                        $\land\ m.acc \in Q\}$
          IN   Check that all acceptors from $Q$ responded
               $\land\ \forall\, a \in Q : \exists\, m \in RMsgs : m.acc = a$
               Check that we have $k$ splits of a chosen value
               $\land\ \exists\, m \in RMsgs :$
                   $\land\ m.chosen \neq None$
                   $\land\ CountSplitsOf(RMsgs, m.chosen) \geq K$
                   $\land\ readLog' = [readLog$ EXCEPT $![b] = m.chosen]$
       $\land$ UNCHANGED $\langle msgs, maxPBal, maxABal, maxVal, chosen \rangle$
    Fast path: Phase 1b quorum has $k$ splits and the value is chosen
    $\lor\ \land\ \exists\, Q \in Quorum1b :$
          LET $RMsgs \triangleq \{m \in msgs :\ \land\ m.type =$ "promise"
                                        $\land\ m.bal\ \ = b$
                                        $\land\ m.acc \in Q\}$
          IN   Check that all acceptors from $Q$ responded
               $\land\ \forall\, a \in Q : \exists\, m \in RMsgs : m.acc = a$
               Check that we have $k$ splits of a chosen value
               $\land\ \exists\, m \in RMsgs :$
                   $\land\ m.chosen \neq None$
                   $\land\ CountSplitsOf(RMsgs, m.chosen) \geq K$
                   $\land\ readLog' = [readLog$ EXCEPT $![b] = m.chosen]$
        $\land$ UNCHANGED $\langle msgs, maxPBal, maxABal, maxVal, chosen \rangle$
     Slow path: Phase 1b recovery and write back
     $\lor\ \land\ \exists\, Q \in Quorum1b :$
           LET $Q1Msgs \triangleq \{m \in msgs :\ \land\ m.type =$ "promise"
                                           $\land\ m.bal\ \ = b$
                                           $\land\ m.acc\ \ \in Q\}$
               $Q1Vals\ \ \triangleq [v\ \ \in Values \cup \{None\} \mapsto$
                                 $\{m \in Q1Msgs : m.maxVal = v\}]$
           IN
              Check that all acceptors from $Q$ responded
              $\land\ \forall\, a \in Q : \exists\, m \in Q1Msgs : m.acc = a$
              $\land\ \exists\, v \in Values :$
                   Check if $v$ is recoverable and of highest ballot
                   Use previous value if $K$ splits exist
```

```
                    ∧ $Cardinality(Q1Vals[v]) \geq K$
                    ∧ $\exists m \in Q1Vals[v]$ :
                        Ensure no other recoverable value has a higher ballot
                        ∧ $\forall mm \in Q1Msgs$ :
                            ∨ $m.bal \geq mm.bal$
                            ∨ $Cardinality(Q1Vals[mm.maxVal]) < K$
                    readLog will be updated in ProposerEnd
                    ∧ $Send([type \mapsto$ “propose”, $bal \mapsto b, val \mapsto v,$
                            $op \mapsto$ “R”$])$
            ∧ UNCHANGED $\langle maxPBal, maxABal, maxVal, chosen, readLog \rangle$
          No value recovered: Return None
          ∨ ∧ $readLog' = [readLog$ EXCEPT $![b] = None]$
            ∧ UNCHANGED $\langle msgs, maxPBal, maxABal, maxVal, chosen \rangle$
```

Next state.

```
Next ≜ ∨ ∃ b ∈ Ballots   : ∨ Prepare(b)
                            ∨ ProposeA(b)
                            ∨ ProposeB(b)
                            ∨ ProposerEnd(b)
                            ∨ FastRead(b)
       ∨ ∃ a ∈ Acceptors : Promise(a) ∨ Accept(a) ∨ Learn(a)
```

$Spec \triangleq Init \wedge \Box[Next]_{vars}$

Invariant helpers.

```
AllChosenWereAcceptedByPhase2 ≜
  ∀ a ∈ Acceptors :
    ∨ chosen[a] = None
    ∨ ∃ Q ∈ Quorum2 :
        ∀ a2 ∈ Q :
          ∃ m ∈ msgs : ∧ m.type = “accept”
                       ∧ m.acc  = a2
                       ∧ m.val  = chosen[a]

OnlyOneChosen ≜
   ∀ a, aa ∈ Acceptors :
      (chosen[a] ≠ None ∧ chosen[aa] ≠ None) ⟹ (chosen[a] = chosen[aa])

VotedForIn(a, v, b) ≜ ∃ m ∈ msgs : ∧ m.type = “accept”
                                   ∧ m.val  = v
                                   ∧ m.bal  = b
                                   ∧ m.acc  = a

ProposedValue(v, b) ≜ ∃ m ∈ msgs : ∧ m.type = “propose”
                                   ∧ m.val  = v
                                   ∧ m.bal  = b
                                   ∧ m.op   = “W”

NoOtherFutureProposal(v, b) ≜
  ∀ vv ∈ Values :
    ∀ bb ∈ Ballots :
      (bb > b ∧ ProposedValue(vv, bb)) ⟹ v = vv

ChosenIn(v, b) ≜ ∃ Q ∈ Quorum2 : ∀ a ∈ Q : VotedForIn(a, v, b)
ChosenBy(v, b) ≜ ∃ b2 ∈ Ballots : (b2 ≤ b ∧ ChosenIn(v, b2))
Chosen(v) ≜ ∃ b ∈ Ballots : ChosenIn(v, b)
```

Invariants.

$LearnInv \triangleq AllChosenWereAcceptedByPhase2 \land OnlyOneChosen$
$ReadInv \triangleq \forall b \in Ballots : readLog[b] = None \lor ChosenBy(readLog[b], b)$
$ConsistencyInv \triangleq \forall v1, v2 \in Values : Chosen(v1) \land Chosen(v2) \implies (v1 = v2)$

$StabilityInv \triangleq$
$\quad \forall v \in Values : \forall b \in Ballots : ChosenIn(v, b) \implies NoOtherFutureProposal(v, b)$

```
$AcceptorInv \triangleq$
  $\forall a \in Acceptors :$
     $\land (maxVal[a] = None) \equiv (maxABal[a] = -1)$
     $\land maxABal[a] \leq maxPBal[a]$
     $\land (maxABal[a] \geq 0) \implies VotedForIn(a, maxVal[a], maxABal[a])$
     $\land \forall c \in Ballots :$
        $c > maxABal[a] \implies \neg\exists v \in Values : VotedForIn(a, v, c)$
```

NetSMC: A Custom Symbolic Model Checker for Stateful Network Verification

Yifei Yuan[1] *Soo-Jin Moon*[2] *Sahil Uppal*[2] *Limin Jia*[2] *Vyas Sekar*[2]
[1]*Intentionet* [2]*Carnegie Mellon University*

Abstract

Modern networks enforce rich and dynamic policies (e.g., dynamic service chaining and path pinning) over a number of complex and stateful NFs (e.g., stateful firewall and load balancer). Verifying if those policies are correctly implemented is important to ensure the network's availability, safety, and security. Unfortunately, theoretical results suggest that verifying even simple policies (e.g., A cannot talk to B) in stateful networks is undecidable. Consequently, any approach for stateful network verification has to fundamentally make some relaxations; e.g., either on policies supported, or the network behaviors it can capture, or in terms of the soundness/completeness guarantees. In this paper, we identify practical opportunities for relaxations in order to develop an efficient verification tool. First, we identify key domain-specific insights to develop a more compact network semantic model which is equivalent to a general semantic model for checking a wide range of policies under practical conditions. Second, we identify a restrictive-yet-expressive policy language to support a wide range of policies including dynamic service chaining and path pinning while enable efficient verification. Third, we develop customized symbolic model checking algorithms as our model and policy specification allows us to succinctly encode network states using existential first-order logic, which enables efficient checking algorithms. We prove the correctness of our approach for a subset of policies and show that our tool, NetSMC, achieves orders of magnitude speedup compared to existing approaches.

1 Introduction

Today's computer networks deploy a large number and variety of complex *stateful* functions [44], ranging from stateful firewalls, NATs, to proxies, and load balancers. Network operators configure those network functions (NFs) to enforce *rich and dynamic policies*, such as dynamic service chaining (e.g., all packets should traverse IPS, while only malicious packets detected by IPS should be sent to FW) [18, 19] and path pinning (e.g., if packets from A to B traverse NF f_1 and then f_2, the reverse packets from B to A should traverse f_2 and then f_1). Formally verifying if the network correctly implements the policies is critical to ensure the availability, security, and safety of the network.

Checking whether policies are correctly enforced, however, is challenging on stateful networks (networks with stateful NFs). Even checking simple policies such as isolation policies (packets from A cannot be delivered to B), an efficiently solvable problem on stateless networks [23, 25–27, 31], is undecidable on stateful networks [47]. In practice, policies enforced on stateful networks will be more complex (see §2).

As such, making practical progress requires non-trivial trade-offs on the supported network behavior, expressiveness of policies, and the soundness/completeness guarantees. For example, recent work VMN [41] simplifies the behavior of a stateful network by assuming that each NF can buffer multiple packets in an out-of-order way, which makes the problem decidable for checking a restrictive set of policies. To verify policies, VMN encodes the network and the policy using first-order logical formulas which are solved by a general-purpose SMT solver. However, it is inefficient to even check the isolation policy due to its high complexity (i.e., EXPSPACE-complete [47]).

In this work, we revisit the stateful network verification problem and explore a different set of relaxation trade-offs in order to achieve more efficient verification for practical scenarios based on the following domain-specific insights:

One-packet at a time network model: Instead of dealing with multiple packets, we adopt a simpler model where only one packet exists in every network state. This model is motivated by the fact that packets inducing conflict behavior on a network are often processed by the network in an *order-preserving* way. For example, connection-based NFs often process packets in a connection in order. Thus, each packet would be processed by the network exactly in the same way when traversing the network with other packets as when traversing alone. Therefore, we can consider only one packet at a time in each network state. While this model simplifies the behavior of a network (e.g., we cannot find violations appearing only under packet interleaving. See §8), we show that the verification result of a wide range of policies (e.g., isolation) based on this model is correct w.r.t. the more complex model in previous work [41] for order-preserving networks (details in §4).

Customized policy and verification algorithm: We design a restricted-yet-expressive policy language based on a subset of linear temporal logic and verification algorithms based on symbolic model checking (SMC) to achieve further speedup of the verification. While our model of network behavior reduces the state space of the problem, checking simple policies (e.g., isolation) efficiently is still hard. Naive approaches based on reducing the problem to constraint solving using general-purpose solvers is not particularly efficient since it would not benefit from the simpler model (see §7).

There are two key challenges to apply the SMC framework to stateful networks: 1) how to succinctly encode a large number of network states and 2) how to efficiently support the computation required in SMC. To this end, we leverage the customized policy structure to use simple existential first order logic (EFO) formulas to succinctly encode a large number of network states. Furthermore, we develop efficient algorithms required in the SMC framework by leveraging the simple network model and extending classic algorithms in other domains (e.g., query containment in database theory).

Based on the key insights discussed above, we implement NetSMC, a symbolic model checker for stateful networks. We prove the correctness of our algorithms w.r.t. a general network semantic model for a subset of policies (e.g., isolation properties). For other policies requiring reasoning about packet interleaving, NetSMC is a sound-but-incomplete bug finding tool that is very efficient. We show the effectiveness of NetSMC via several use cases using real-world NFs such as pfSense [3] and HAProxy [2] running in Cloudlab [43]. We evaluate NetSMC on various network topologies and policies and show that NetSMC scales to networks with hundreds of stateful NFs and is $>$ 200X faster than the state-of-the-art stateful network verification tool VMN [41] on typical fattree-topology networks.

2 Motivation

We motivate the stateful network verification problem by describing several practical policies, followed by the key challenges of the verification problem.

2.1 Stateful Network Verification Examples

Isolation. Consider a network (Fig. 1a) with a stateful firewall to protect the Department from the Internet. Network operators may enforce the isolation policy: traffic from untrusted hosts in the Internet cannot be sent to the Department.
Conditional reachability. Continuing the example above, the network operators may additionally enforce the policy to allow all traffic from the Department and to allow traffic from those trusted hosts in the Internet that have a connection already established from a host in Department.
Flow affinity. Consider a load balancer that distributes traffic among n servers as shown in Fig. 1b. To keep the service provisioning undisrupted, the network operator wants to enforce the following flow affinity policy: if a packet from a host Client is load balanced to a server, then all future packets in the same flow should always be sent to the same server.
Dynamic service chaining. Fig. 1c shows a multi-stage intrusion prevention system (IPS) consisting of a light IPS and a heavy IPS. Each device in the network is configured such that all traffic from the Department is sent to the light IPS, which performs basic detection such as counting the number of bad connections for each host. If a host is detected suspicious by light IPS (e.g. issuing more than 10 bad connections), all future packets from the host should be directed to the heavy IPS for further processing; otherwise its traffic is directly sent to the Internet.
Path pinning. Often a network needs to deploy multiple instances of the same middlebox function for better throughput.

	Buzz & Symnet	VMN	**NetSMC**
Model	One-packet	Out-of-order	One-packet
Policy lang.	Assertion	LTL-based	LTL-based
Correctness	Sound	Sound, Complete	Sound, Cond. complete

Table 1: Comparison with network verification tools.

Consider the network shown in Fig. 1d which is configured to forward traffic between the Department and the Internet to one of the firewalls. An interesting path pinning policy is: if a packet from H1 in the Department to H2 in the Internet goes through the i-th firewall, then all future packets from H2 to H1 should traverse the same firewall.

2.2 Challenges

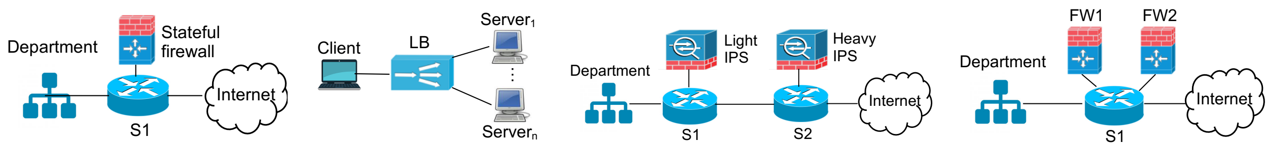
Stateful network verification is more challenging compared to stateless verification. In general, this problem has been shown to be undecidable even for simple isolation policies (see Theorem 1 in [47]). As such, any practical progress needs to make practical relaxations on at least one of the following dimensions: the supported network behavior, the expressiveness of policies, and the correctness guarantees.

As an example, VMN [41] recovers the decidability of the problem by assuming that an NF buffers packets in an out-of-order fashion instead of FIFO. VMN reduces the verification problem to encoding network configurations and policies as first-order logical constraints which can be solved by an SMT solver. Since solving general first-order constraints is undecidable, VMN targets policies in the form of "if packet p reaches node B then p must not satisfy some property P in the past", so that the encoded constraints fall in a decidable fragment. Even with the above simplification the verification problem induces high complexity (i.e., EXPSPACE-hard [47]). Thus, VMN is not scalable to large-size networks even for checking simple isolation policies as shown in §7.

3 Overview

The undecidability result [47] means that *any* approach in this space has to seek some relaxations or tradeoffs in order to make the problem tractable. One of our contributions is identifying and exploring a different point in this space of practical relaxation choices in order to develop an efficient verification tool in practical network scenarios. Table 1 summarizes key difference of our trade-offs compared with existing work.

More specifically, our trade-off is based on the following two key domain-specific insights: First, the key challenge to stateful network verification is to handle interleaving among multiple packets in the network (e.g., packet p_1 is processed by NF A before packet p_2, but p_2 is processed by NF B before p_1). In practice, however, networks often exhibit intrinsic *order-preserving* property in several scenarios, where packets that induce conflict network behavior are processed in the same order. For example, a syn packet is often processed by the network before a synack packet is sent into and processed by the network. Motivated by this observation, we



(a) Stateful firewall. (b) Load balancing. (c) Multi-stage IPS. (d) Multiple stateful firewalls.

Figure 1: Examples

adopt a semantic model that only allows one packet being considered at each network state. In effect, we consider a sequential execution model of networks, similar to the ones considered in stateful network testing tools [18, 45]. Second, even with this simple model verifying simple isolation policies is still hard (i.e. PSAPCE-hard) and naively employing a general-purpose tool is not efficient for large-size networks (see §7). Motivated by recent success in customized stateless network verification tools (e.g., HSA [26], VeriFlow [27]), we design a customized policy language and verification technique for stateful networks based on the symbolic model checking (SMC) framework. We are able to identify efficient symbolic representations for network states and develop efficient SMC algorithms.

Next, we discuss our key design choices before delving into more detailed algorithmic designs in the following sections.

Network model (§4). To model the behavior of a stateful network, we need two key components, an NF model that can capture various NF behavior and the modeling of packets traversing the network as we discussed above.

We need an expressive yet restrictive model that can model various NF behavior while supporting efficient verification. On one end of the spectrum, we could use a general purpose language (e.g., C in Buzz [18]), but that leads to highly inefficient verification. Motivated by existing NF models [7, 52], a stateful NF can be modeled as: 1) a set of state tables indexed by packet header fields, and 2) a set of rules that modify those state tables based on packet matching and table testing results of the incoming packet. Such restricted formalization enables efficient checking algorithms in SMC as we show in §6.

Policy specification language (§5). As shown in §2, stateful network policies have temporal properties, so a temporal specification language such as linear temporal logic (LTL) [42] is a natural choice for specifying such policies. While it is expressive enough for a wide set of network policies, the full set of LTL is computationally difficult to handle.

Our insight is that the set of network policies of interest fall into the intersection of LTL and computational tree logic (CTL) which has more efficient verification algorithms [13]. Therefore, we identify a subset of LTL that is intuitive and expressive to specify a wide range of policies while using efficient verification algorithms based on CTL. As we show in §6, reasoning about policies in this set of policies also enables us to use succinct symbolic encoding of network states and efficient checking algorithms required in SMC.

Efficient verification algorithm (§6). We design customized verification techniques based on classic SMC framework for efficiency. Algorithms for the symbolic model checking framework are well known. However, there are a set of challenges to instantiate the framework in the context of stateful network verification. First, we need a succinct symbolic representation for a large set of network states, particularly for the internal state (e.g., connection state tables) of NFs. Second, the symbolic model checking algorithm requires computing the pre-image of a symbolic state (i.e., the set of states that can transition to this state in one step), and the termination of the symbolic model checking framework requires efficient computation to check containment of two sets of states in the symbolic representation. How to efficiently support the computations remains another challenge. While classic data structures such as BDD are widely used in other contexts, it is not suitable in the context of stateful networks due to the large state space. For example, a stateful firewall maintains state for each flow and thus the number of states is as large as 2^F where F, the maximal number of flows a firewall tracks, can be in the order of thousands. Our design of the network model and policies allows us to address those two challenges. First, we succinctly encode a large set of states into a fragment of existential first-order logic (EFO). As an example for a stateful firewall, we may use $\exists x, y. Trust[x, y] = 1$ to represent all network states where there is a legitimate flow (a src-dst pair for simplicity) recorded by the connection table *Trust*. Our choice of the policy language ensures that any symbolic state emerged during the computation of SMC can be encoded in EFO. Second, our simple NF model allows us to efficient compute the pre-image of a symbolic state . Moreover, using EFO, we identify the connection between containment checking of sets of states in EFO and conjunctive query containment problem in the database community [11, 29]. We adapt the query containment checking algorithm to efficiently check the containment of two sets of network states.

4 Stateful Network Model

We present our stateful network model, including the NFs and semantic rules. We illustrate the expressiveness of our NF model via example encodings of common network functions.

4.1 NF Model

We summarize the syntax in Fig. 2. Inspired by prior work [7, 52], each NF includes local state which are key-value maps and a set of rules for processing packets and updating local state. NFs' computations are restricted to state checking, simple counting, and non-deterministic value choice. This model is more expressive than the one used by the verification

Field Name	f	$\in$	{`srcip`, `dstip`, `srcport`...}
Value	v	$\in$	$\mathrm{Int} \cup \mathrm{IP} \cup ...$
Packet	pkt	$::=$	$\{\overrightarrow{f_i = v_i}\}$
Location	l	$\in$	Loc
Located Packet	lp	$::=$	(l, pkt)
State Table	T	$\in$	TableNames
Expression	e	$::=$	$v \mid f \mid \mathbf{pickFrom}(D) \mid T[\overrightarrow{e_i}]$
Atomic Test	at	$::=$	$\mathrm{True} \mid \mathtt{loc} \in D \mid f \in D \mid T[\overrightarrow{e_i}] \in D$
Test	t	$::=$	$at \mid \neg at \mid t,t$
Update	u	$::=$	$T[\overrightarrow{e_i}] := e \mid \mathbf{inc}(T[\overrightarrow{e_i}], v) \mid \mathbf{dec}(T[\overrightarrow{e_i}], v)$
Action	a	$::=$	$\mathbf{fwd}(e) \mid \mathbf{drop} \mid \mathbf{modify}(f, e)$
Command	c	$::=$	$u \mid a \mid c;c$
Rule	r	$::=$	$t \Rightarrow c$
NF Config	R	$::=$	$\cdot \mid r; R$
NF	NF	$::=$	$(\overrightarrow{l}, \overrightarrow{T}, R)$
Network Topo	$topo$	$\in$	$\mathrm{Loc} \to \mathrm{Loc}$
Network Config	N	$::=$	$(topo, [NF_1, .., NF_k])$
Table Valuation	Δ	$\in$	$\mathrm{TableNames} \to \delta_T$
Network State	s	$::=$	(lp, Δ)

Figure 2: Syntax of stateful network model.

tool VMN [41] and is efficient (§6).

Basics. We write *pkt* to denote packets, which are records of packet fields (notation $\{\overrightarrow{t_i}\}$ is a shorthand for $\{t_1, \cdots, t_n\}$). Packet field names, denoted f, are drawn from a set of predefined names, including common field names such as `srcip`, `dstip`, `srcport` and user-defined application specific field names. We use Loc to denote the set of all locations (e.g., interfaces at a switch) in the network, including two special ones: Drop (denoting that packets are dropped) and Exit (denoting that packets exit the network). A located packet, denoted *lp*, is a pair of a location and a packet.

NF. We model all the network devices as *network functions*, denoted *NF*, which is a tuple consisting of a set of locations $\overrightarrow{l}$ (i.e. interfaces), a set of tables $\overrightarrow{T}$ for storing internal state (e.g. a stateful firewall may use state tables to store connection state), and a list of rules R that process packets and update its state. Stateless devices' state tables are empty.

A rule r consists of a list of tests on packet fields and state tables, denoted t, and a sequence of commands, denoted c, for updating the state and generating the outgoing packet. For instance, a stateful firewall may drop or forward the packet (captured by c) depending on the result of testing the packet headers and the internal state (captured by t). A rule r is fired, i.e., its commands are executed, when the current packet and state tables pass the tests in r.

We allow the following atomic tests: trivial tests that return true; tests that check the current location of the incoming packet; tests that check whether a field value or the value of a state table entry is in a specified finite domain D (e.g., an interval). Common features such as longest prefix matching for IP addresses can be modeled using $f \in D$. A command c is a sequence of updates to state tables, denoted u and actions applied to packets, denoted a.

We write e to denote expressions that can be used by rules of NFs, which include constants, packet field values indexed by field names, values picked (nondeterministically) from a domain D (**pickFrom**(D)), and values stored in state tables. Each state table is a finite key-value map, where we write $T[\overrightarrow{e_i}]$ to denote the value in an entry indexed by the key $\overrightarrow{e_i}$.

A state table can be updated. The update $T[\overrightarrow{e_i}] := e$ updates the entry indexed by the key $\overrightarrow{e_i}$ to the value of e. We allow simple counting operations to model IDS/IPS. $\mathbf{inc}(T[\overrightarrow{e_i}], v)$ increments the value in the table entry by a constant v; $\mathbf{dec}(T[\overrightarrow{e_i}], v)$ performs decrementing similarly. We consider the following actions for incoming packets: forwarding, dropping, and modifying the value of a packet field. We do not model multicasting or broadcasting in this paper.

Upon receiving a packet *pkt* at a location l, an NF attempts to match the located packet $lp = (l, pkt)$ with all of its rules. The matching succeeds if all atomic tests in the rule are true given *lp* and the current state tables. For an atomic test that involves a field name f (e.g., $f \in D$), that field name evaluates to the value of the field f in the packet *pkt*. As an example, an atomic test `Trust[`**`dst,src`**`]=1` first evaluates **`src`** and **`dst`** to be the source and destination addresses of the incoming packet *pkt*, then uses the concrete values as the key to look up the entry in the table `Trust`, and finally checks if the corresponding entry is `1`. If the matching succeeds, all actions and updates of the rule are applied sequentially. Without loss of generality, we assume that exactly one rule can match an incoming packet. It is straightforward to translate other models such as the one based on first-match into this model.

4.2 NF Examples

To demonstrate the expressiveness of our model, we show example encodings of several stateful network functions. Writing a NF model is a one-time effort, and can be automated (c.f. [48]), which is out of the scope of this work.

Stateful firewall. A stateful firewall protects an internal network by restricting accesses from external hosts. Fig. 3 shows the code snippet of a stateful firewall. Here, we assume that the internal network is connected to location 0 of the stateful firewall, and the outside network is connected to location 1. The stateful firewall uses a state table `Trust` to keep track of the flows that are established by the internal network. Initially, all entries in `Trust` have value 0. When a packet comes from the internal network, the firewall forwards it directly to the outside, and updates the state table entry for that flow to 1 (the 1st rule). When a packet comes from outside (location 1), the firewall first checks the state table to see whether a packet in the reverse direction has been seen (i.e., the table entry is 1); if so, the packet is forwarded (the 2nd rule); otherwise the packet is dropped (the 3rd rule).

Load balancer. A load balancer forwards packets destined for a virtual destination of a service (e.g., online searching) to one of the backend servers that implement the service. Fig. 4 shows a load balancer for a service with virtual IP address

```
loc=0 => Trust[src,dst]:= 1, fwd(1);
loc=1, Trust[dst,src]=1 => fwd(0);
loc=1, Trust[dst,src]=0 => drop;
```

Figure 3: Stateful firewall.

VIP, where we assume that servers are connected to location 1 and clients are connected to location 0. The load balancer maintains two state tables, `Connected` for storing whether a client has been assigned to a server and `Server` for storing the address of the server assigned to each client. Initially all table entries have value 0, indicating that no server has been assigned to any client. The first rule corresponds to the case where a client was assigned to a server (i.e. `Connect[src]=1`), and the load balancer needs to modify the destination address of the packet to the address of the assigned server as stored in the `Server` table. Similarly, the second rule accounts for the case where the client has not been assigned to any servers (i.e., `Connect[src]=0`). In this case, the load balancer picks a server from all the backend servers `D`, updates the state tables, and modifies the packet destination accordingly. Note that the use of `pickFrom(D)` abstracts away the concrete mechanism of choosing the server for a client. For packets not destined to the service, the load balancer may drop the packets as shown in the third rule. Last, for traffic going from servers to clients, the load balancer simply modify the source address of the packet to the virtual address, as indicated by the last rule.

```
loc=0, dst=VIP, Connected[src]=1 =>
  modify(dst, Server[src]), fwd(1);
loc=0, dst=VIP, Connected[src]=0 =>
  Server[src]:=pickFrom(D), Connected[src]:=1,
  modify(dst, Server[src]), fwd(1);
loc=0, dst!=VIP => drop;
loc=1 => modify(src, VIP), fwd(0);
```

Figure 4: Load balancer.

4.3 Network Semantic Model

Network configuration. We consider a network configuration N as a set of links, denoted *topo*, together with a set of NFs in the network. We model the execution of a stateful network as a state transition system, where each state corresponds to a snapshot of the network (referred to network state) and a transition between two network states denotes an atomic step of feasible network execution. In each network state, we need to consider the valuation of state tables of each NF as well as the packets in the network. We use a function Δ, which maps each state table T to a function δ_T, to denote the valuation of all NF state tables.

Packet processing in the network. To model packets in the network, a general approach is to associate each NF interface with *an infinite* FIFO queue buffering packets to be processed. As mentioned in §1, unfortunately, policy checking in such models is undecidable. Recent progress [41] relaxes this model by assuming packets are buffered in an out-of-order way. While this out-of-order model recovers decidability for some policies, it still incurs high computational complexity.

For efficient verification, we consider a model where only one packet exists in any network state. We model a network state as a pair (lp,Δ), where lp is a located packet being processed and Δ is a table valuation. Our model has three types of transitions: (1) At a network state (lp,Δ), the packet is received by a *NF* and *NF* modifies the located packet to lp' and updates the state tables to Δ', and the network evolves to state (lp',Δ'); (2) When a packet pkt is moved from one end l of a link to the other end l', a network state $((l,pkt),\Delta)$ can evolve to $((l',pkt),\Delta)$; (3) When the current packet pkt is dropped or exits the network, a new packet at an ingress location is brought into the network.That is, $((O,pkt),\Delta)$ can evolve to $((I,pkt'),\Delta)$, where O is Drop or Exit, I denotes an ingress location, and pkt' is an arbitrary packet. We write $E = s_0 \xrightarrow{lp_0/lp_1} \cdots \xrightarrow{lp_{n-1}/lp_n} s_n$ to denote execution traces, where s_i is a network state, lp_{i-1}/lp_i denotes the processed located packet and the resulting located packet in step i respectively. We assume no indefinite loops for any packet; transient loops are allowed. Detailed rules are in Appendix A.

Connecting to packet-interleaving model. Our one-packet model excludes packet interleaving behaviors, and thus cannot find all policy violations. To answer the question: *when can packet interleaving be safely ignored*, we first formalize when do the two models agree and what do they agree on.

Let us write $\mathsf{E}^{\infty}(N)$ to denote the set of all closed (i.e., both the initial and final states have only packets at locations Drop or Exit) finite execution traces of the network N under the packet interleaving semantics; similarly, we write $\mathsf{E}^{\mathsf{one}}(N)$ to denote the set of all finite closed network execution traces under the one-packet model. We assume that each packet has an unique ID, that is not modified by any NFs. Given a network execution trace $E = s_0 \xrightarrow{lp_0/lp_1} \cdots \xrightarrow{lp_{n-1}/lp_n} s_n \in \mathsf{E}^{\mathsf{one}}$ of a network N and a packet ID id, we define *per-packet trace* for a packet with ID id, denoted $E|_{id}$, as the sequence $[lp_{i_1}/lp_{i_1+1} \dots lp_{i_k}/lp_{i_k+1}]$ obtained by keeping only those lp_i/lp_{i+1} pairs whose packet ID is id. Per-packet trace for $E \in \mathsf{E}^{\infty}$ can be defined similarly.

It turns out that most network policies are checkable on per-packet traces. For example, checking the policy "packets from N1 cannot reach N2" can be achieved by examining packet-traces for every packet in the network. Then, when are the per-packet traces of the one-packet model the same as the per-packet traces of the packet interleaving model?

Our key insight is that a wide range of network scenarios are intrinsically *order-preserving*, where packets whose interleaving matters (i.e., swapping their process orders induces divergent network behavior) are processed in the same order by all NFs in the network. For example, the state update of connection-based stateful NFs is triggered by control packets of a connection, which are often sent to and processed by the network in order (e.g., a syn packet is processed before a synack packet). Thus, a network with connection-based stateful NFs is order-preserving. As another example, modern network devices often integrate a pipeline of NFs where packets traverse them in order. Networks with such stateful devices on the edge of the network such that any packet only traverses one of them is also an order-preserving network. The formal definitions are in Appendix C. We can show that for order-preserving networks, the packet interleaving model and the one-packet model agree on per-packet traces. Formally:

Lemma 1 *Given an order-preserving network N, $\forall E \in \mathsf{E}^{\infty}(N)$, $\exists E' \in \mathsf{E}^{\mathsf{one}}(N)$, s.t. $\forall id$, $E|_{id} = E'|_{id}$.*

The above lemma is the building block for proving the conditional soundness and completeness results of our algorithm.

5 Network Policies

To strike a reasonable balance between efficiency and expressiveness, we use a subset of the linear temporal logic (LTL) as our specification language. This language is expressive enough to specify a wide range of policies as we show in §7 and also is more efficient for policy checking compared to the full set of LTL formulas. Policies in this language can be translated to equivalent ones in the computational tree logic (CTL), thus allowing more efficient checking algorithms of CTL to be used [14]. Further, a simple fragment of first order logic can be used to reason about network states (details in §6). We envision tools could be used to build policy templates (e.g., [41]) or GUI (e.g., [18]), to ease the policy specification process. We provide a high-level overview of our policy specification language with details in Appendix B. We show an example policy specification from §2. We end by a theorem stating that the one-packet and packet interleaving model agree on checking several practical policies.

Syntax. The syntax of our specification language is shown below. Predicates, denoted θ, include equality checks between a packet field value, the current location of the packet, and a state table value and a variable (a symbolic value).

Predicate	θ	::=	$f = x \mid \mathtt{loc} = x \mid T[\overrightarrow{e_i}] = x$
Basic formula	γ	::=	$\theta \mid \neg\gamma \mid \gamma_1 \wedge \gamma_2 \mid \gamma_1 \vee \gamma_2$
Temporal formula	ρ	::=	$\gamma \mid \mathtt{F}\,\gamma \mid \mathtt{f}\,\gamma \mid \mathtt{X}\,(\gamma \to \rho)$
			$\mid \mathtt{G}\,(\gamma \to \rho) \mid \mathtt{g}\,(\gamma \to \rho)$
Policy	P	::=	$\overrightarrow{\forall x_i \in D_i}.\rho$

We write γ to denote basic formulas, which include predicates and propositional connectives. A temporal formula is denoted ρ, whose semantics is defined on an execution trace E, assuming the first state of E is the current state. An execution trace E satisfies $\mathtt{F}\gamma$ if γ is true on some future network state in E. We do not have the $\mathtt{U}$ operator in LTL but introduce two special operators $\mathtt{f}$ and $\mathtt{g}$ to specify properties that should hold during the traversal of the current packet in the network. More concretely, $\mathtt{f}\gamma$ is the short-hand for $(\mathtt{loc} \neq \mathsf{Drop} \wedge \mathtt{loc} \neq \mathsf{Exit})\,\mathtt{U}\gamma$. E satisfies $\mathtt{f}\gamma$ if γ holds on some future network state in the current packet's traversal. $\mathtt{g}\gamma$ is the short-hand for $\gamma\mathtt{U}(\mathtt{loc} = \mathsf{Drop} \vee \mathtt{loc} = \mathsf{Exit})$. It is true on E if γ is true on every network state in the current packet's traverse. Nested temporal formulas is only allowed in the restricted forms: $\mathtt{X}(\gamma \to \rho)$, $\mathtt{G}(\gamma \to \rho)$ and $\mathtt{g}(\gamma \to \rho)$, where $\mathtt{X}(\gamma \to \rho)$ states that starting on the next state γ is true entails ρ is true, and $\mathtt{G}(\gamma \to \rho)$ ($\mathtt{g}(\gamma \to \rho)$, resp.) intuitively asserts that whenever γ holds (during the traversal of the current packet, resp.) ρ should also hold. A network policy is a closed temporal formula, universally quantified at the outermost layer. A network N satisfies a policy P, denoted $N \models P$ iff for all execution E starting from an initial network state of N, E satisfies P. Formally definitions are in Appendix B.

The predicates here are customized to one-packet model in that a packet field name uniquely identifies the packet in the network state. By prefixing each packet field with a packet ID (i.e., $id.f$), we can express (equivalent) policies for networks with packet interleaving.

An example. We show the specification of the dynamic service chaining policy in §2. More examples can be found in §7. For space constraint, we consider a sub-policy of the dynamic service chaining policy: if a host is detected suspicious by Light IPS then all of its future packets should be directed to Heavy IPS. Suppose Light IPS keeps an internal state table named *susp* which counts bad connection numbers from each host, and a host is suspicious when the count is larger than 10. The top-level structure of the policy is $\forall x.\mathtt{G}(\gamma(x) \to \rho(x))$, which specifies that for all host x whenever $\gamma(x)$ holds $\rho(x)$ should also hold. Here, $\gamma(x) = (\mathtt{src} = x \wedge susp[x] > 10)$ specifies that the *susp* count of x is larger than 10 and the current packet is from x; $\rho(x)$ specifies that whenever a packet is sent from x, it will eventually reach H (i.e. the location of Heavy IPS) before being dropped or exiting the network. We can specify this policy as follows.

$$\forall x \in Dept. \quad \mathtt{G}(\mathtt{src} = x \wedge susp[x] > 10 \to \\ \mathtt{G}(\mathtt{src} = x \to \mathtt{f}(\mathtt{loc} = H)))$$

Model equivalence. We write $N \models^{\infty} P$ to denote that a network N satisfies policy P in the packet interleaving model; we use $N \models^{one} P$ to denote that N satisfies P in the one-packet model. A policy P^{one} specified for the one-packet model can be translated to a policy for the packet-interleaving model, denoted P^{∞}. For example, the translated isolation polity in Appendix F is $\forall id.\mathtt{G}(id.loc = A \to ((id.loc \neq B)\mathtt{U}(id.loc = \mathsf{Drop} \vee id.loc = \mathsf{Exit}))$. We prove the following theorem stating that checking a number of policies is not affected by ignoring the packet interleaving (Appendix D).

Corollary 2 *For all order-preserving networks N, $N \models^{\infty} P^{\infty}$ if and only if $N \models^{one} P^{one}$, if P is the isolation, tag preservation, or tag-based isolation policy.*

6 NetSMC Checking Algorithms

We view the verification of policies on a network as a model checking problem. Our choice of the policy language in the

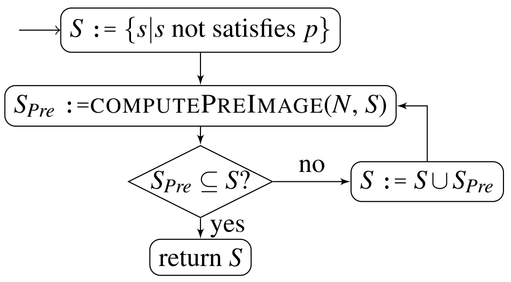

$S := \{s | s \text{ not satisfies } p\}$
S_{Pre} :=COMPUTEPREIMAGE(N, S)
$S_{Pre} \subseteq S$?
no
$S := S \cup S_{Pre}$
yes
return S

Figure 5: Generic SMC algorithm workflow to check G p

restricted LTL form allows us to use the more efficient CTL model checking algorithm. In addition, to handle the large state space we adopt the symbolic model checking (SMC) framework for CTL. In this section, we first provide some background of SMC to highlight key challenges in implementing an SMC tool, and then discuss our approaches to address those challenges.

6.1 Background on SMC

Symbolic model checking is a verification technique that has been proven to be effective and efficient in many domains [14, 35]. Given a system model N and a policy P to be checked, a generic SMC framework computes the set of system states S that violates P (i.e. the set of states satisfying $\neg P$). Then the algorithm checks whether an initial state is in the set S. If so, a violation of the policy P is found; otherwise, P is verified.

To compute the set of states S satisfying $\neg P$, an SMC algorithm computes the set of states that satisfies each sub-formula of $\neg P$ bottom-up, following the structure of $\neg P$. As an example, Figure 5 shows the generic SMC algorithm to compute the set of states that violates G p, i.e., from those states there exists an execution trace of the system that violates p.

Initially, the algorithm computes the set of states violating the sub-formula p. Then it repeatedly adds states that can reach some state violating p. In each iteration of the loop, the algorithm computes the set of states S_{Pre} that can transition to a state in S in one step (i.e., $S_{Pre} = \{s | \exists s' \in S.s \rightarrow s'\}$, called the pre-image of S). The algorithm converges when every state in S_{Pre} is contained in S, and then returns the desired set.

To enable efficient SMC, we need a succinct symbolic encoding for a large set of states (e.g. S and S_{Pre}) while supporting efficient computation over them as shown in pre-image computation and subset checking. In the following, we present key components of our algorithm to address these challenges.

6.2 Symbolic Network States in EFO

To illustrate the intuition of our symbolic encodings, consider the firewall example in Fig. 3, where we are interested in checking whether packets from external networks can reach the internal network. Based on the firewall model, a packet from the outside is only allowed to go through if the tests in the second rule return true. The tests return true for all network states where an entry with value 1 exists in the table `Trust`. That is, the network states of interest can be encoded as $\exists x, y.Trust[x,y] = 1$. Thanks to our policy specification, any set of states generated in the checking of (violation of) a policy can be encoded in such an existential form (see details later in this section). Therefore, we can use the following fragment of existential first-order logic (EFO) as our symbolic state encoding. The existential quantifications are only at the outermost level (we thus omit quantifiers) and we operate on the inner formulas without quantifiers.

Atomic Predicates	α	$::=$	$x \in D \mid x \neq y$
		$\mid$	$loc = x \mid f = x \mid T[\overrightarrow{x_i}] = y$
Clauses	β	$::=$	$\bigwedge_i \alpha_i$
State Formulas	ϕ	$::=$	$\bigvee_i \beta_i$

A set of network states is encoded using a state formula, written ϕ, in a DNF form. We say that a network state (lp, Δ) is encoded by ϕ, if there exists a substitution of concrete values for all free variables in ϕ such that (lp, Δ) satisfies ϕ under that substitution. We write $Sat(\phi)$ to denote the set of network states encoded by ϕ. The atomic predicates, α, include membership predicate, inequality check, and test for fields, location and state tables. For the firewall example above, the state formula ϕ is $(Trust[x,y] = 1)$, encoding all network states where the firewall has an entry in *Trust* with value 1. As we show in the following, the encoding of EFO enables efficient computation of key components in SMC.

6.3 Computing Pre-Image

Next, we describe how to compute the state formula of the pre-image of a symbolic state S, denoted $Pre(S)$, (i.e. COMPUTEPREIMAGE). That is, given a state formula ϕ, we need to compute the state formula ϕ_{Pre}, such that $Sat(\phi_{Pre}) = Pre(Sat(\phi))$. We develop an algorithm that directly generates ϕ_{Pre} by transforming ϕ based on the network model.

Notation. Before explaining the algorithm, we define some auxiliary notations. Without loss of generality, we assume that for each clause β in ϕ, each field f appears at most once (any formula can be rewritten to this form). We write $var(f, \beta)$ to denote the variable being compared to f in β. That is, $var(f, \beta) = x$ if $f = x$ appears in β. If f does not appear in β, $var(f, \beta)$ returns a fresh variable. We write $\beta \backslash_\alpha$ to denote the formula resulted from removing the clause α from β.

Top-level algorithm. Our pre-image computing algorithm (shown in Alg. 1) considers all three types of network transitions (c.f. §4). The top-level function COMPUTEPREIMAGE takes as inputs the network model and the state formula ϕ and returns ϕ_{Pre}. The loop (lines 2-5) goes over every rule in every network function to generate a pre-image that could reach ϕ using that rule. Function COMPUTEPRERULE accounts for the network transitions under NF processing; Function COMPUTELINK on line 6 computes the pre-image for link traversal; Function COMPUTELASTPKT on line 7 computes the pre-image when ϕ represents the state where a new packet

Algorithm 1 Computing the pre-image of a state formula.

```
 1: function COMPUTEPREIMAGE($N$, $\phi$)
 2:     for all NF in the network do
 3:         $(L, \overrightarrow{T}, R) \leftarrow \texttt{NF}$
 4:         $\phi_{\texttt{NF}} := \bigvee_{l \in L, r \in R} \bigvee_{\beta \in \phi_r} ((\texttt{loc} = l) \wedge \beta)$
 5:         where $\phi_r$ := COMPUTEPRERULE($r$, $\phi$)
 6:     $\phi_{link}$ := COMPUTELINK($\phi$)
 7:     $\phi_{pkt}$ := COMPUTELASTPKT($\phi$)
 8:     return $\bigvee_{\texttt{NF}} \phi_{\texttt{NF}} \vee \phi_{pkt} \vee \phi_{link}$
 9: function COMPUTEPRERULE($r$, $\phi$)
10:     $t \Rightarrow c \leftarrow r$
11:     $\phi_c$ := COMPUTEPRECMD($c$, $\phi$)
12:     return $\bigvee_{\beta \in \phi_c} (\bigwedge_{at \in t} trans(at) \wedge \beta)$
13: function COMPUTEPRECMD($c$, $\phi$)
14:     match $c$ with
15:         | $a \Rightarrow$ return COMPUTEPREACTION($a$, $\phi$)
16:         | $u \Rightarrow$ return COMPUTEPREUPDATE($u$, $\phi$)
17:         | $c_1, c_2 \Rightarrow \phi_2$ := COMPUTEPRECMD($c_2$, $\phi$)
18:             return COMPUTEPRECMD($c_1$, $\phi_2$)
19: function COMPUTELASTPKT($\phi$)
20:     $\phi'$:=False
21:     for all $\beta$ in $\phi$ do
22:         $\beta' := \beta \backslash_{\texttt{loc}=var(\texttt{loc},\beta)}$
23:         for all $f = x$ in $\beta$ do
24:             $\beta' := \beta' \backslash_{f=x}$
25:         for all ingress location $l$ do
26:             $\beta_1 := (var(\texttt{loc}, \beta) = l) \wedge \beta' \wedge (\texttt{loc} = \text{Drop})$
27:             $\beta_2 := (var(\texttt{loc}, \beta) = l) \wedge \beta' \wedge (\texttt{loc} = \text{Exit})$
28:             $\phi' := \phi' \vee \beta_1 \vee \beta_2$
29:     return $\phi'$
```

Algorithm 2 Sub-functions of computing the pre-image.

```
 1: function COMPUTEPREACTION($a$, $\phi$)
 2:     match $a$ with
 3:         | fwd($e$) $\Rightarrow (g_e, x_e) := F(e)$
 4:             return $\bigvee_{\beta \in \phi} \beta \backslash_{\texttt{loc}} \wedge g_e \wedge (var(\texttt{loc}, \beta) = x_e)$
 5:         | drop $\Rightarrow$
 6:             return $\bigvee_{\beta \in \phi} \beta \backslash_{\texttt{loc}} \wedge (var(\texttt{loc}, \beta) = \text{Drop})$
 7:         | modify($f$, $e$) $\Rightarrow (g_e, x_e) := F(e)$
 8:             return $\bigvee_{\beta \in \phi} \beta \backslash_{f} \wedge g_e \wedge (var(f, \beta) = x_e)$
 9: function COMPUTEPREUPDATE($u$, $\phi$)
10:     T[$\overrightarrow{e_i}$]:=$e \leftarrow u$
11:     $(g_{e_i}, x_{e_i}) := F(e_i)$ for all $e_i$
12:     $(g_e, x_e) := F(e)$
13:     $g := \bigwedge_i g_{e_i} \wedge g_e$
14:     for all $\beta_j$ in $\phi$ do
15:         $\phi_j$ := COMPUTECLS($u$, $\beta_i$, $[(g_{e_i}, x_{e_i})]$, $(g_e, x_e)$)
16:     return $\bigvee_i \bigvee_{\beta \in \phi_i} g \wedge \beta$
17: function COMPUTECLS($u$, $\beta$, $[(g_{e_i}, x_{e_i})]$, $(g_e, x_e)$)
18:     let tList be the list of state tests $T(\overrightarrow{x}) = y$ in $\beta$
19:     match tList with
20:         | nil $\Rightarrow$ return $\beta$
21:         | $h$::$hs \Rightarrow \phi_0 =$ COMPUTECLS($u$, $\beta \backslash_h$)
22:             T[$\overrightarrow{e_i}$]:=$e \leftarrow u$, $(T(\overrightarrow{x}) = y) \leftarrow h$
23:             $\beta_0 := (\overrightarrow{x} = \overrightarrow{x_{e_i}}) \wedge (y = x_e)$
24:             $\beta_j := h \wedge (x_j \neq x_{e_j})$ for $j = 1, .., m$
25:             return $\bigvee_{\beta' \in \phi_0} ((\beta_0 \wedge \beta') \vee \bigvee_j (\beta_j \wedge \beta'))$
```

enters the network. The algorithm returns the disjunction of all formulas for each possible transition. Note that the returned state formula is still in EFO, which enables us to only consider EFOs for state containment. Next, we describe two key functions. We omit the third as it is similar.

Packet transitions. Function COMPUTELASTPKT computes the pre-image when a new packet comes to the network. It computes the pre-image of each clause β in ϕ, and returns the disjunction of all computed pre-images. If a network state $((l', pkt'), \Delta)$ is the result of the transition, then the state before this transition has the same state tables but a different packet. Therefore, all constraints on packet fields and locations are removed from β (line 22-24) as they do not apply to the packet in the pre-image. Furthermore, the location of the packet in the pre-image must be either Drop or Exit, and l' must be an ingress location. Thus, constraints $\texttt{loc} = \text{Drop}$, $\texttt{loc} = \text{Exit}$ are added, the same for $var(loc, \beta) = l$ for each ingress location l (line 26, 27).

NF transitions. The function COMPUTEPRERULE iteratively computes the pre-image ϕ_c under the actions and updates in r (line 9 in Alg. 1). The pre-image under rule r is obtained by adding constraints of the tests t in r using the helper function *trans* (not shown due to space) which translates each atomic test into a clause. Key sub-routines are summarized in Alg. 2.

Function COMPUTEPREACTION computes the pre-image of an action a. Consider the case where a is **modify**(f, e). The semantics of **modify** require the value of field f be modified to the value of e. Thus, the algorithm first considers the value returned by the expression e using the helper function F, which returns a clause g_e together with a variable x_e given expression e. The intuitive meaning is that if g_e is satisfied, then the value of e is equal to x_e (F's formal definitions is omitted). As an example for $e = (T[\texttt{src}, \texttt{dst}])$, $g_e = (\texttt{src} = y_1 \wedge \texttt{dst} = y_2 \wedge T[y_1, y_2] = y_3)$, and $x_e = y_3$. Then the algorithm adds the constraint g_e and $var(f, \beta) = x_e$. Furthermore, since the value for f is modified, the constraints associated with f from β can be removed as they do not apply to the pre-image.

Function COMPUTEPREUPDATE computes the pre-image under an update $T[\overrightarrow{e_i}] := e$; **inc** and **dec** are similar. The function computes the pre-images for each clause β using the sub-procedure COMPUTECLS, and returns the union of them. Function COMPUTECLS recursively enumerates all possible effects of u to β to compute its pre-image. More concretely, the function considers two cases that u may impact a constraint $T(\overrightarrow{x}) = y$ in β: 1) $T(\overrightarrow{x}) = y$ is updated by u, and 2) $T(\overrightarrow{x}) = y$ is not updated by u. In the first case, it must be the cases that $\overrightarrow{x} = \overrightarrow{x_{e_i}}$ and $y = x_e$, where x_{e_i} denotes the value read from e_i (β_0 shown in line 23). In the second case, $x_i \neq x_{e_i}$ for at least one x_i (line 24). We obtain the pre-image of β as a disjunction of the two case shown in line 25.

6.4 Containment of Network States

As shown in Figure 5, we need an efficient approach to check the containment of two sets of network states. Given two state formulas ϕ_1 and ϕ_2, we need to check if $Sat(\phi_1) \subseteq Sat(\phi_2)$. This is equivalent to checking $\exists \overrightarrow{x}.\phi_1 \Rightarrow \exists \overrightarrow{y}.\phi_2$, where $\overrightarrow{x}$ ($\overrightarrow{y}$, resp.) denotes all free variables in ϕ_1 (ϕ_2, resp.). While this can be solved using a general-purpose SMT solver, as we show in the evaluation section, this is quite inefficient.

Instead, we observe that the state containment problem of EFO is a variant of the *query containment* problem well-studied in database theory [11]. In short, query containment aims to determine if the result of a database query q_1 is contained in that of q_2 for all database instances I. To make the connection clearer, consider the state formula $\phi_1 = (\texttt{src} = x \wedge \texttt{dst} = y \wedge Trust[y,x] = 1)$. This formula can be viewed as the following (conjunctive) query on a database with three tables: *src*, *dst* and *Trust*. $q_1(x,y) : -src(x), dst(y), Trust(y,x)$

Each concrete network state can be viewed as a database instance with the schema defined by packet fields and state tables. Furthermore, each state formula ϕ is a union of conjunctive queries, where each clause β in ϕ is a conjunctive query with inequalities between variables.

In database theory, to determine whether a conjunctive query q_1 is contained in another conjunctive query q_2, it is equivalent to checking whether there is a *homomorphism* from q_2 to q_1, i.e. a function h that maps variables in q_2 to variables and constants in q_1, such that for all $R(x_1,x_2,..)$ in q_2, there is an $R(h(x_1),h(x_2),..)$ in q_1 [11].

However, there are still a few challenges in applying the algorithm to our problem. First, as shown in [29], when there are inequalities, there may not exist a homomorphism even when ϕ_1 is contained in ϕ_2. For example, $\phi_1 = (x_1 \neq x_3 \wedge T[x_1,x_2] = 1 \wedge T[x_2,x_3] = 1)$ is contained in $\phi_2 = (y_1 \neq y_2 \wedge T[y_1,y_2] = 1)$, but there is not homomorphism from ϕ_2 to ϕ_1. Second, in query containment problem a variable ranges over a continuous domain (e.g. rational numbers) [29], while in network verification a variable can only take discrete values such as IP addresses. As a result again, there may not exist a homomorphism, even if a set of states encoded in ϕ_1 is contained in the set of states encoded in ϕ_2, E.g., $\phi_1 = (x \in \{0\} \wedge y \in \{0\} \wedge T_1[x] = 1 \wedge T_1[y] = 0)$ is contained in $\phi_2 = (T_2[z] = 0)$ since no states are encoded by ϕ_1, but no homomorphism exists.

To address the first challenge, we break each clause in ϕ_1 into *atomic clauses*, which has been shown to handle inequalities [29]. We call a clause β an atomic clause w.r.t. a state formula ϕ_2, if all variables in β are distinct, and for all variables x in β and y in ϕ_2, the domain of x is either contained in or disjointed with the domain of y. For the first example above, ϕ_1 can be break into the following three atomic clauses, namely, $\beta_1 = (x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_1 \neq x_3 \wedge T[x_1,x_2] = 1 \wedge T[x_2,x_3] = 1)$, $\beta_2 = (x_1 \neq x_2 \wedge T[x_1,x_2] = 1 \wedge T[x_2,x_2] = 1)$, and $\beta_3 = (x_1 \neq x_3 \wedge T[x_1,x_1] = 1 \wedge T[x_1,x_3] = 1)$. We see that there is a homomorphism from ϕ_2 to β_3. For the second challenge, we check for emptiness of clauses, and show that given a state formula ϕ_2 and an atomic clause β w.r.t. ϕ_2, $Sat(\beta) \subseteq Sat(\phi_2)$ if and only if there is a homomorphism from some $\beta' \in \phi_2$ to β, or $Sat(\beta)$ is empty. Now we can verify the containment in the second example above. We obtain our algorithm of checking containment by putting these two pieces together (Alg. 3).

Algorithm 3 Checking containment

```
1: function CHECKCMT($\phi_1$, $\phi_2$)
2:     for all $\beta$ in $\phi_1$ do
3:         let $[\beta_0,..,\beta_k]$ be the set of atomic clauses w.r.t. $\phi_2$ obtained from $\beta$
4:         for all $i = 0$ to $k$ do
5:             if ISEMPTY($\beta_i$) then continue
6:             if there is no homomorphism from $\beta'$ to $\beta_i$ for all $\beta' \in \phi_2$ then
7:                 return False
8:     return True
```

We prove the correctness of our algorithm w.r.t. the one-packet semantics: if NetSMC says policy verified, then all possible executions of the network satisfy the policy; and if NetSMC says policy violated, then there exists an execution that violates the policy. Formally:

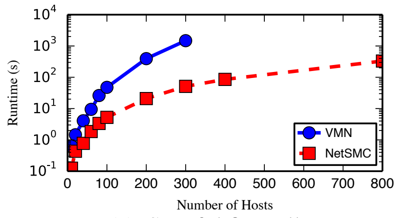
Theorem 3 (Correctness) *Given a stateful network N and a policy P, NetSMC returns True if and only if N satisfies the policy P under the one-packet model.*

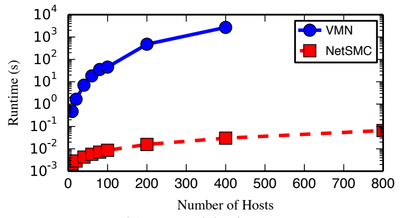
The above theorem uses our one-packet semantics. Combined with theorems like Lemma 1 , the verification results of our tool on a large set of practical scenarios are correct w.r.t. the general packet interleaving semantic model as well.

7 Evaluation

We implement a prototype tool NetSMC in Python based on the algorithms above. We evaluate NetSMC and show that:

- NetSMC can scale to large-size networks and is orders of magnitude more efficient than existing approaches (§7.1);
- Our custom algorithm on containment checking in NetSMC is effective and is 42 times more efficient than naive approaches based on general-purpose solvers (§7.1);
- NetSMC can check a wide range of network policies in various practical network scenarios, which can not be easily supported in alternative tools (§7.2).


(a) Stateful firewall


(b) Load balancer

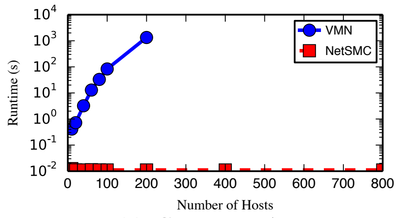
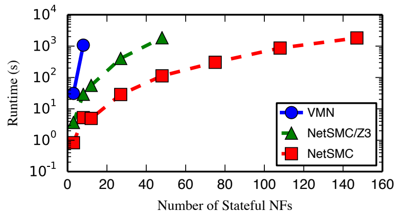
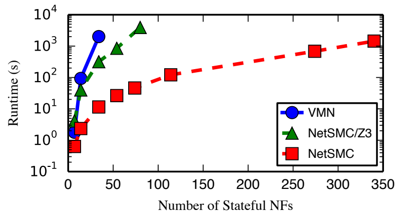

(c) Content cache

Figure 6: Scalability with NF complexity


(a) Fattree


(b) Ai3

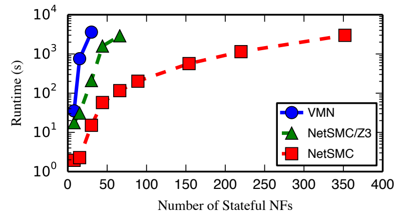

(c) Sprint

Figure 7: Scalability with network complexity.

Evaluation setup: We ran NetSMC on a server with 20 cores (2.8GHz) and 128GB RAM. We first evaluate NetSMC's scalability by varying the complexity of NFs, topologies, and policies (§7.1). For comparison, we use the open-source implementation of VMN [41], a state-of-the-art stateful network verification tool. We also demonstrate the effectiveness and expressiveness in a range of network scenarios using real NFs based on emulation in Cloudlab [43] (§7.2). We use pfSense [3] as stateful firewalls and NATs, HAProxy [2] as load balancers. For the case study that required dynamic rule installation (e.g., path pinning), we used the Mininet-based emulation with POX [34] as the SDN controller. We ensure high-fidelity of NetSMC NF models using Alembic [38], which can automatically synthesize NF models from NF implementations. For NFs that are not readily available from Alembic (e.g., POX programs), we manually translate NF programs into equivalent NetSMC models.

7.1 Scalability

NF complexity. Stateful NFs may implement complex functionalities using multiple configuration rules. To evaluate the scalability of NetSMC w.r.t. the complexity of NFs, we consider three types of stateful NFs: (1) a stateful firewall, (2) a load balancer, and (3) a content cache. To create NF configurations with varying complexity, we connect n hosts and n servers to each NF, and for each pair of hosts and servers, we add a rule to the NF. For example, for the stateful firewall, we add rules to limit access from servers to hosts.

Fig. 6 shows the runtime on verifying isolation of a server to a host. NetSMC is orders of magnitude faster than VMN on all tested NFs. Particularly, for the stateful firewall experiment, VMN takes 1477 seconds to verify the policy with 300 hosts, while NetSMC only takes 51 seconds (28×). In the load balancer experiment, VMN takes 2693 seconds with 400 hosts while NetSMC only takes 0.03 seconds. We observe similar speedup in the cache experiment.

Topology complexity. We consider the fattree [4] topology and Ai3 and Sprint, from Topology Zoo [30]. For fattree, we create a range of topologies by varying the number of ports per switch. For Ai3 and Sprint, we systematically extend each switch with multiple switches to generate topologies with varying size. For each topology with n switches, we add additional $2n/3$ stateful NFs where each switch is attached to at most one stateful NF. We use each tool to verify the isolation policy of two hosts in each network. Since VMN critically relies on the slicing technique, we slice the flowspace of all tested networks before applying both tools.

Fig. 7 shows the runtime of verification tools w.r.t. the number of *stateful* NFs in the network. We make the following observations. First, NetSMC is at least two orders of magnitude faster than VMN. Specifically, VMN spends 1072 seconds on the fattree network with 8 stateful NFs, while NetSMC only uses 5 seconds (200× faster). Furthermore, VMN cannot scale to larger networks within 12 hours, while NetSMC can successfully verify the desired policy for networks with 147 stateful NFs in half an hour. For Ai3 and Sprint network, we see similar performance speedup. For example, VMN uses 2011 seconds on the Ai3 network with 34 stateful NFs while NetSMC only uses 11 seconds (175×).

Effectiveness of customized algorithm. To evaluate the benefit of our custom algorithms, we consider an alternative approach by using Z3 to solve the containment problem of network states in NetSMC (shown as NetSMC/Z3 in Fig. 7). We observe that our custom algorithm on containment checking significantly improves the scalability. When using Z3 to check containment, the tool uses 1844 seconds to verify the policy for the fattree network with 48 stateful NFs, which is 16× slower than our custom approach. On Ai3 and Sprint, we measured 42× and 25× speedup respectively.

Policy complexity. To evaluate the scalability of NetSMC w.r.t. the complexity of the policy to be checked, we use NetSMC to check a range of service chaining policies with varied number of NFs on the chain. Since VMN's implementation does not support this type of policy, we consider the variant of NetSMC that uses Z3 for containment checking for comparison. Fig. 8 plots the results. First, we observe that NetSMC can scale up to reasonably large policies. Particularly, NetSMC can check the service chaining policy with 20 NFs in 20 minutes. Second, we observe again that our custom algorithm on containment checking significantly improves the performance: on 12 NFs, our custom model checking algorithm is 23× faster than the Z3 variant.

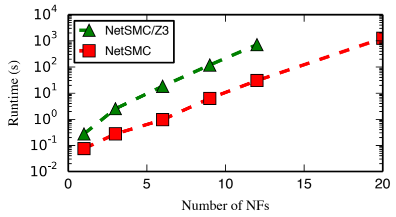


Figure 8: Scalability with policy complexity.

Comparison with general-purpose model checkers. To evaluate the benefit of our custom symbolic model checking algorithm, we further compare NetSMC with a classical BDD-based symbolic model checker NuSMV [12] and a SMT-based model checker Cubicle [15]. Since NuSMV cannot effectively model the state tables using small BDD structures, we model state tables with fixed sizes in NuSMV. We repeat the stateful firewall experiment as described above. With table size 16, NuSMV takes 1163 seconds on verify the reachability policy, while NetSMC only uses 0.015 seconds without size constraint on the state tables. NetSMC is 750X faster than Cubicle. The result confirms that our encoding of symbolic states and custom algorithms for stateful networks are more efficient than general-purpose encodings and tools.

7.2 Effectiveness and Expressiveness

Red-blue team exercise: To validate the effectiveness of NetSMC, we conduct a red-blue team exercise in a range of network scenarios using real NFs. In each scenario, the red team (Author 2 and Author 3) set up a network with intended policies in CloudLab and then delele/modify NF rules (which are kept secret from the blue team) to introduce misconfiguration. The blue team (Author 1) uses NetSMC to check intended policies on the network, so as to identify and fix the misconfiguration.

• **Blocking hosts behind NAT [19]:** The red team sets up a network with two subnetworks (N1 and N2) and two NFs using pfSense with the intended policy to block a host h1 in N1 from reaching another host h2 in N2. However, using NetSMC, the blue team identifies a violation that packets from h1 can still reach h2. The root cause is that the red team mistakenly adds a NAT rule in the first NF such that h1's address is translated and bypassing the firewall rule installed on the second NF blocking h1's address. The blue team fixes this misconfiguration by adding the firewall rule on the first NF and NetSMC verifies the policy.

Policy & network scenario	Time: Verification	Time: Bug find
Conditional reach.: A stateful firewall with ACL rules.	0.06s	0.03s
Data isol. [41]: A content cache with a client and a server.	2.23s	0.0007s
Pipeline [41]: A stateful firewall with two hosts and servers.	0.001s	0.0006s
Flow affinity: As described in Fig. 1b.	0.19s	0.04s
Dynamic service chaining: As described in Fig. 1c.	0.1s	0.008s
Reachability: Two cascaded NATs [1] (Outside can reach the inside server).	0.001s	0.005s
Tag-based isol.: Network as in Fig. 1c. (A packet labeled by a specific tag should not pass a specific MB).	0.04s	0.94s
Tag preservation: Network as in Fig. 1c. (A packet's tag labeled by a MB should be not be modified).	0.03s	0.98s
NAT consistency: If a NAT modifies a packet's port then all future packets in the flow should have the same port.	0.09s	0.078s

Table 2: Example policies supported by NetSMC.

• **Opposite rules in firewalls:** The red team sets up a network with two subnetworks (N1 and N2) and two stateful firewalls in each subnetwork (fw1 in N1 and fw2 in N2) to protect the subnetworks. The intended policy is to allow N1's packet to reach N2. The blue team uses NetSMC to check this policy and find a violation that packets from N1 is allowed by fw1 but denied at fw2. The blue team fixes the misconfiguration by removing the rule blocking N1 on fw2. Using the new model after the fix, NetSMC successfully verifies the policy.

• **Consistent load balancing:** The red team configures HAProxy to enforce the policy that packets from the same host should always be load balanced to the same server. Using NetSMC, the blue team finds a violation where one flow is sent to server 1 while another is sent to server 2. Checking the configuration, the blue team identifies that HAProxy is misconfigured in the "round robin" mode thus violating the policy. The blue team then reconfigures the LB in the "Source" mode. NetSMC successfully verifies the desired policy.

• **Path pinning:** The red team sets up the network scenario in Fig. 1d with two hosts for the Department and Internet. To enforce the path pinning policy, the red team uses a POX controller to program the forwarding rules on s1. The blue team uses NetSMC to check the path pinning policy, which finds a violation where the first packet from Department is sent to FW1 but the return packet is sent to FW2. The root cause is that the controller mistakenly installed a wrong rule on the switch for the return packets. The blue team fixes the problem by using consistent rules on the controller. Then, NetSMC then successfully verifies the policy.

Policy expressiveness. We show a wide range of policies that can be specified and checked using NetSMC, summarized in Table 2. For each case, we simulate networks in NetSMC as described in the table and introduce misconfiguration by deleting/modifying rules in NFs. NetSMC identifies the misconfiguration in all cases and can verify all policies after fixing the bugs. We report the time of verifying the policy or finding bugs. We can see that NetSMC significantly expands the scope of efficiently verifiable network policies. Today's stateless verification tools cannot model any network scenarios considered in the table, and existing stateful network verification tools [5, 41] cannot specify some of the policies (summarized in Appendix D).

8 Limitation and Discussion

One-packet model: NetSMC is built on top of the one-packet model, and thus may not find violations caused by packet interleaving. For instance, in the second example of the red-blue team exercise, if fw1 drops all packets from N1 once receiving packets from N2, while fw2 allows packets from N1 to reach N2 after seeing packets from N2 to N1, then NetSMC declares packets from N1 cannot reach N2. However, the following violating trace is missed by the one-packet model: first, a packet p1 from N1 passes fw1, then fw2 processes packet p2 from N2 to N1. Next, fw2 allows p1 to reach N2.

NF Model: NetSMC only supports header matching, state table checking, and simple counting; more complex operations, such as computing average values, are not supported.

Policy: Our policy language cannot express policies needing arbitrary quantification, nesting of temporal operators, or generic until path formulas. For instance, the policy that "at some time in the future, a packet from h1 to h2 is delivered", $\texttt{F}(loc = h1 \rightarrow \texttt{f}(loc = h2))$, is beyond our scope.

Network failures: Currently NetSMC does not model network failures directly and can only check policies in the presence of failure by enumerating each failure scenario and run NetSMC in each case, which may not be efficient.

9 Related Work

Our stateful network model is motivated by existing work on network modeling and programming languages [6, 7, 21, 28, 36, 37, 52]. Our NF model shares key characteristics with the models in NetEgg [52] and SNAP [7].

There is a rich body of work for testing and verifying forwarding behaviors in stateless networks [23, 25–27, 31, 32, 46, 49, 50, 53, 53, 54]. While those work can efficiently check a number of policies such as reachability and loop freedom, it is nontrivial to extend those work to support stateful data planes, which are the target of our work.

For example, Header Space Analysis (HSA) [26] models each packet as a point in the high-dimension space of packet headers and each switch as a transfer function from a subspace into another. Based on symbolic reasoning of the transfer functions, HSA can efficiently verify policies such as reachability and loop freedom. Adapting HSA for stateful network verification would need to introduce some notion of state to the transfer function, which would require a complete redesign of the verification algorithm.

Veriflow [27] uses an alternative "trie" like encoding and focuses on checking policies incrementally when network configurations change. Whenever a rule change occurs on a switch, Veriflow computes the packet space that is influenced by the change, and only applies verification to the delta part. Again, adding state to the trie structure and its associated algorithms is non-trivial.

NoD [31] is based on a generic Datalog framework to check reachability policies, where both networks and policies are encoded in Datalog. NoD can potentially be extended to model stateful network functions. However, Datalog is limited in its expressive power in terms of network policies; temporal properties such as flow affinity and dynamic service chaining cannot be easily specified. It is also unclear if such a stateful extension (if exists) scales.

Our work is closely related to recent efforts on stateful data plane testing and verification. Buzz [18] and SymNet [45] generate test cases for stateful networks based on symbolic execution. VMN [41] verifies isolation properties based on SMT encodings. Alpernas et al. present abstractions to check isolation properties [5]. Those projects, except VMN, only support a subset of our policies. We cannot express generic policies involving past operations; while VMN cannot express some policies NetSMC supports. Our approach is also different in the network model and we build a highly custom symbolic model checker to improve the efficiency.

There are several complementary proposals on verifying control planes: Batfish [20], ERA [17], ARC [22] and Minesweeper [9] analyze routing control planes. NICE [10], VeriCon [8], SDNRacer [16], FlowLog [39] and Kuai [33] target SDN controllers. Work on verifying firewalls [24, 40, 51, 55] can handle statefulness in firewalls, but it is not clear whether those techniques can generalize to handle more expressive network functions and policies.

10 Conclusions

This paper explores a different design space in building efficient verification tools for stateful networks. We identify key domain-specific insights to define a compact model of stateful networks, customize policy specifications, and develop efficient custom symbolic model checking algorithms for verification. We implement NetSMC and show that it achieves orders of magnitude speedup compared to alternative approaches, while supporting a wide range of policies.

Acknowledgements: This work is supported in part by the National Science Foundation via grant CNS1513961 and CNS1552481, and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] What is Double NAT? https://kb.netgear.com/30186/What-is-Double-NAT.

[2] haproxy. https://www.haproxy.org/.

[3] pfSense. https://www.pfsense.org/.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, 2008.

[5] Kalev Alpernas, Roman Manevich, Aurojit Panda, Mooly Sagiv, Scott Shenker, Sharon Shoham, and Yaron Velner. Abstract Interpretation of Stateful Networks. *arXiv preprint arXiv:1708.05904*, 2017.

[6] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 2014.

[7] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, 2016.

[8] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 31. ACM, 2014.

[9] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, 2017.

[10] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford, et al. A nice way to test openflow applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[11] Ashok K Chandra and Philip M Merlin. Optimal Implementation of Conjunctive Queries in Relational Data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*. ACM, 1977.

[12] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, 2002.

[13] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, Berlin, Heidelberg, 1982. Springer-Verlag.

[14] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[15] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems . In *Proceedgins of International Conference on Computer-Aided Verification (CAV)*, 2012.

[16] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. SDNRacer: concurrency analysis for software-defined networks. In *ACM SIGPLAN Notices*, volume 51, pages 402–415. ACM, 2016.

[17] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[18] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. Buzz: Testing context-dependent policies in stateful networks. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[19] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, 2014.

[20] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. A General Approach to Network Configuration Analysis. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI*, 2015.

[21] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David

Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291. ACM, 2011.

[22] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[23] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time Network Verification Using Atoms. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[24] Alan Jeffrey and Taghrid Samak. Model checking firewall policy configurations. In *Proceedings of IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, 2009.

[25] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[26] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[27] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[28] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[29] Anthony Klug. On conjunctive queries containing inequalities. *Journal of the ACM (JACM)*, 35(1):146–160, 1988.

[30] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[31] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking Beliefs in Dynamic Networks. In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[32] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, 2011.

[33] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *Proceedgins of Formal Methods in Computer-Aided Design (FMCAD)*, 2014.

[34] J Mccauley. Pox: A python-based openflow controller, 2014.

[35] Kenneth L McMillan. Symbolic Model Checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[36] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices*, 47(1):217–230, 2012.

[37] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[38] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. Alembic: Automated model inference for stateful network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[39] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[40] Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave Tool for Firewall Analysis. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, 2010.

[41] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying Reachability in Networks with Mutable Datapaths. In *Proceedings of 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[42] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, Oct 1977.

[43] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[44] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, 2012.

[45] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, 2016.

[46] Brendan Tschaen, Ying Zhang, Theo Benson, Sujata Benerjee, JK Lee, and Joon-Myung Kang. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *Proceedings of IEEE SDN-NFV Conference*, 2016.

[47] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. Some complexity results for stateful network verification. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2016.

[48] Wenfei Wu, Ying Zhang, and Sujata Banerjee. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, 2016.

[49] Geoffrey G Xie, Jibin Zhan, David A Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2015.

[50] H. Yang and S. S. Lam. Real-Time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, April 2016.

[51] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and Prasant Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, 2006.

[52] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. Scenario-based Programming for SDN Policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, CoNEXT '15, 2015.

[53] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. *IEEE/ACM Trans. Netw.*, 22(2):554–566, April 2014.

[54] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[55] Shuyuan Zhang, Abdulrahman Mahmoud, Sharad Malik, and Sanjai Narain. Verification and synthesis of firewalls using SAT and QBF. In *Proceedings of 20th IEEE International Conference on Network Protocols (ICNP)*, 2012.

Appendix

A Semantics of Stateful Network Model

The top-level transition rules are of the form: $s \rightarrow s'$. We use a number of auxiliary transitions summarized below:

$[\![e]\!]_{lp;\Delta} = v$	Expression evaluation
$[\![at]\!]_{lp;\Delta} = b$	Atomic test evaluation
$[\![t]\!]_{lp;\Delta} = b$	Test evaluation
$c; lp; \Delta \rightarrow lp'; \Delta'$	Command evaluation
$r; lp; \Delta \rightarrow lp'; \Delta'$	Rule evaluation
$\texttt{NF}; lp; \Delta \rightarrow lp'; \Delta'$	NF evaluation

The semantic rules of our stateful network model are summarized in Figure 10, Figure 11, and Figure 9.

$\boxed{s \rightarrow s'}$

$$\textsc{Net-Trans-NF}\quad \frac{NF = (L,\_,\_) \qquad l \in L \qquad lp = (l, pkt) \qquad NF;(l,pkt);\Delta \rightarrow (l',pkt');\Delta'}{(lp,\Delta) \rightarrow ((l',pkt'),\Delta')}$$

$$\frac{topo(l) = l'}{((l,pkt),\Delta) \rightarrow ((l',pkt),\Delta)}\ \textsc{Net-Link}$$

$$\frac{l = \mathsf{Drop/Exit} \qquad l' \in IngressLocs}{((l,\_),\Delta) \rightarrow ((l',pkt),\Delta)}\ \textsc{Net-Packet}$$

Figure 9: One-packet semantics of network execution.

$\boxed{[\![e]\!]_{lp;\Delta} = v}$

PICKFROM
$$\frac{v \in D}{[\![\mathbf{pickFrom}(D)]\!]_{lp;\Delta} = v}$$

FIELD
$$\frac{lp = (\_, pkt) \qquad pkt.f = v}{[\![f]\!]_{lp;\Delta} = v}$$

STATE-TABLE
$$\frac{\forall i, [\![e_i]\!]_{lp;\Delta} = v_i \qquad \delta_T = \Delta(T) \qquad \delta_T(\overrightarrow{v_i}) = v}{[\![T[\overrightarrow{i = e_i}]]\!]_{lp;\Delta} = v}$$

$\boxed{[\![at]\!]_{lp;\Delta} = b}$

TEST-LOC-TRUE
$$\frac{lp = (l, \_)}{[\![\mathtt{loc} = l]\!]_{lp;\Delta} = \mathsf{True}}$$

TEST-LOC-FALSE
$$\frac{lp = (l', \_) \qquad l \neq l'}{[\![\mathtt{loc} = l]\!]_{lp;\Delta} = \mathsf{False}}$$

TEST-FIELD-TRUE
$$\frac{[\![f]\!]_{lp,\Delta} \in D}{[\![f \in D]\!]_{lp;\Delta} = \mathsf{True}}$$

TEST-FIELD-FALSE
$$\frac{[\![f]\!]_{lp,\Delta} \notin D}{[\![f \in D]\!]_{lp;\Delta} = \mathsf{False}}$$

TEST-TABLE-TRUE
$$\frac{[\![T[\overrightarrow{i = e_i}]]\!]_{lp;\Delta} = v}{[\![T[\overrightarrow{i = e_i}] = v]\!]_{lp;\Delta} = \mathsf{True}}$$

TEST-TABLE-FALSE
$$\frac{[\![T[\overrightarrow{i = e_i}]]\!]_{lp;\Delta} = u \qquad u \neq v}{[\![T[\overrightarrow{i = e_i}] = v]\!]_{lp;\Delta} = \mathsf{False}}$$

TEST-NEG
$$\frac{}{[\![\neg at]\!]_{lp;\Delta} = \neg[\![at]\!]_{lp;\Delta}}$$

$\boxed{[\![t]\!]_{lp;\Delta} = b}$

TEST-SEQUENCE-TRUE
$$\frac{[\![t_1]\!]_{lp;\Delta} = \mathsf{True}}{[\![t_1, t_2]\!]_{lp;\Delta} = [\![t_2]\!]_{lp;\Delta}}$$

TEST-SEQUENCE-TRUE
$$\frac{[\![t_1]\!]_{lp;\Delta} = \mathsf{False}}{[\![t_1, t_2]\!]_{lp;\Delta} = \mathsf{False}}$$

Figure 10: Semantics of network function.

$\boxed{c; lp; \Delta \rightarrow lp'; \Delta'}$

UPDATE
$$\frac{\begin{array}{c}[\![e]\!]_{lp;\Delta} = v \qquad \forall i, [\![e_i]\!]_{lp;\Delta} = v_i \\ \delta_T = \Delta(T) \qquad \delta'_T = \delta_T[\overrightarrow{v_i} \mapsto v] \qquad \Delta' = \Delta[T \mapsto \delta'_T]\end{array}}{T[\overrightarrow{i = e_i}] :\!= e; lp; \Delta \rightarrow lp; \Delta'}$$

INC
$$\frac{\begin{array}{c}\forall i, [\![e_i]\!]_{lp;\Delta} = v_i \qquad \delta_T = \Delta(T) \\ \delta'_T = \delta_T[\overrightarrow{v_i} \mapsto \delta_T(\overrightarrow{v_i}) + v] \qquad \Delta' = \Delta[T \mapsto \delta'_T]\end{array}}{\mathbf{inc}(T[\overrightarrow{i = e_i}], v); lp; \Delta \rightarrow lp; \Delta'}$$

DEC
$$\frac{\begin{array}{c}\forall i, [\![e_i]\!]_{lp;\Delta} = v_i \qquad \delta_T = \Delta(T) \\ \delta'_T = \delta_T[\overrightarrow{v_i} \mapsto \delta_T(\overrightarrow{v_i}) - v] \qquad \Delta' = \Delta[T \mapsto \delta'_T]\end{array}}{\mathbf{dec}(T[\overrightarrow{i = e_i}], v); lp; \Delta \rightarrow lp; \Delta'}$$

ACTION-FORWARD
$$\frac{lp = (l, pkt) \qquad [\![e]\!]_{lp;\Delta} = v \qquad lp' = (v, pkt)}{\mathbf{fwd}(e); lp; \Delta \rightarrow lp'; \Delta}$$

ACTION-DROP
$$\frac{}{\mathbf{drop}; (l, pkt); \Delta \rightarrow (\mathsf{Drop}, pkt); \Delta}$$

ACTION-MODIFY
$$\frac{lp = (l, pkt) \qquad [\![e]\!]_{lp;\Delta} = v \qquad pkt' = pkt[f \mapsto v]}{\mathbf{modify}(f, e); lp; \Delta \rightarrow (l, pkt'); \Delta}$$

SEQUENCE
$$\frac{c_1; lp; \Delta \rightarrow lp'; \Delta'}{(c_1; c_2); lp; \Delta \rightarrow c_2; lp'; \Delta'}$$

$\boxed{r; lp; \Delta \rightarrow lp'; \Delta'}$

RULE
$$\frac{[\![t]\!]_{lp;\Delta} = \mathsf{True} \qquad c; lp; \Delta \rightarrow lp'; \Delta'}{t \Rightarrow c; lp; \Delta \rightarrow lp'; \Delta'}$$

$\boxed{\mathtt{NF}; lp; \Delta \rightarrow lp'; \Delta'}$

NF
$$\frac{\begin{array}{c}lp = (l, pkt) \\ l \in L \qquad r_j \in R \qquad r_j; lp; \Delta \rightarrow lp'; \Delta'\end{array}}{(L, \overrightarrow{T_j}, R); lp; \Delta \rightarrow lp'; \Delta'}$$

Figure 11: Semantics of network function (cont.).

B Policy Semantics

We define the semantics of open formulas ρ over a tuple $(\mathtt{V}, E, N)$ (Figure 2), where $\mathtt{V}$ is the valuation function of all free variables appearing in ρ, E is an infinite network execution trace (i.e., a sequence of network states), and N is the network configuration. We write E_i to denote the i-th state in E and $E[i..]$ as the suffix of E starting at the i-th state, Similar to the semantic rules, we omit the network configuration N for simplicity of presentation. $(\mathtt{V}, E)$ satisfying ρ, written $(\mathtt{V}, E), \models \rho$ is formally defined below.

- $(\mathtt{V}, E) \models \gamma$ iff $E_0 \models \gamma$ in the standard way.
- $(\mathtt{V}, E) \models \mathtt{G}\rho$ iff $(\mathtt{V}, E[i..]) \models \rho$ for all $i \geq 0$
- $(\mathtt{V}, E) \models \mathtt{g}\rho$ iff there is an $i \geq 0$ where $E_i = (lp, \Delta)$, s.t. 1) $lp = (\mathsf{Drop}, \_)$ or $lp = (\mathsf{Exit}, \_)$ and 2) for all $j < i$, $(\mathtt{V}, E[j..]) \models \rho$.
- $(\mathtt{V}, E) \models \mathtt{F}\gamma$ iff $(\mathtt{V}, E[i..]) \models \gamma$ for some i
- $(\mathtt{V}, E) \models \mathtt{f}\gamma$ iff there is an $i \geq 0$ s.t. 1) $E_i \models \gamma$ and 2) for all $j < i$ where $E[j] = (lp, \Delta)$, $lp \neq (\mathsf{Drop}, \_)$ and $lp \neq (\mathsf{Exit}, \_)$
- $(\mathtt{V}, E) \models \mathtt{X}\rho$ iff $(\mathtt{V}, E[1..]) \models \rho$.

C One packet vs. packet interleaving model

We identify sufficient conditions under which our one packet model is equivalent to the interleaving model considered in [41] w.r.t. a set of policies. First, we give the formal semantics of the interleaving model; then we show sufficient conditions under which our one packet model is equivalent to the interleaving model, followed by our proofs establishing the equivalence.

Executions in packet interleaving network model: A network state in the packet interleaving model is a pair $(\mathtt{Q},\Delta)$, where $\mathtt{Q}$ is the set of packets buffered at each network location and Δ is the valuation function of state tables as usual. The network state of the one-packet model is a special case where $|\mathtt{Q}| = 1$. We use $ID(p)$ to denote the ID of packet p and $ID(lp)$ has its natural meaning. The top-level transition rules of the packet interleaving model are given in Fig. 12, where the three rules generalized the corresponding rules in the one packet model. Each transition in the transition system of the packet-interleaving model is of the form of $s \xrightarrow{lp/lp'}_{NF} s'$, where lp/lp' denote the (located) packet that is processed by the transition and lp (lp', resp.) denote the packet before (after, resp.) the transition, NF denotes the NF that processes that packet or *null* (which we typically ignore) if the packet is transmitted by a link or injected into the network.

A network execution trace E is a sequence of transitions $s_0 \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_n$. A *closed network execution trace* is a finite network execution where the both initial state s_0 and the final state s_n contain no packets buffered at any location except for Drop and Exit. We use $\mathsf{E}^\infty(N)$ to denote the set of all closed network execution traces of a given network N under the packet interleaving semantics; similarly, we use $\mathsf{E}^{\mathsf{one}}(N)$ to denote the set of all closed network execution traces under the one-packet model. We assume that there is no indefinite loops for any packet traversal; transient loops are allowed to appear.

Formalization of processing-order preserving: We give necessary definitions first before we formalize the processing-order preserving condition.

Definition 4 (Non-conflict) *Given NF, lp_1, lp_2, we say that lp_1 is non-conflicting with lp_2 at NF if $\forall \Delta_0, \Delta_1, \Delta_2, lp'_1, lp'_2$ s.t. $NF; lp_1; \Delta_0 \to lp'_1; \Delta_1$ and $NF; lp_2; \Delta_1 \to lp'_2; \Delta_2$, $\exists \Delta'_1$ s.t. $NF; lp_2; \Delta_0 \to lp'_2; \Delta'_1$ and $NF; lp_1; \Delta'_1 \to lp'_1; \Delta_2$.*

We define processing-order preserving based on packet orders. Particularly, given a set of packets, a packet order $\prec$ is a strict total order on the IDs of the packets. Given two located packets lp_1 and lp_2, we denote $lp_1 \prec lp_2$ if $ID(lp_1) \prec ID(lp_2)$.

Definition 5 (Processing-order preserving trace) *Given a network N, a network execution trace $E = s_0 \Rightarrow \ldots \Rightarrow s_n$ (under the packet interleaving model) in $\mathsf{E}^\infty(N)$, a packet order $\prec$, E is* processing-order preserving *(or order-preserving in short) under $\prec$ if $\forall lp_1, lp_2, NF$ such that $lp_2 \prec lp_1$ and lp_1 is conflicting with lp_2 at NF, there do not exist transitions $s_j \xrightarrow{lp_1/lp'_1}_{NF} s_{j+1}$ and $s_k \xrightarrow{lp_2/lp'_2}_{NF} s_{k+1}$ in E where $k \geq j+1$.*

$$\boxed{(\mathtt{Q},\Delta) \xrightarrow{lp/lp'}_{NF} (\mathtt{Q},\Delta')}$$

NET-TRANS-NF

$$\frac{\begin{array}{c} NF = (L,\_,\_) \\ l \in L \qquad pkt \in \mathtt{Q}(l) \qquad NF;(l,pkt);\Delta \to (l',pkt');\Delta' \\ \mathtt{Q}' = \mathtt{Q}[l' \mapsto (\mathtt{Q}(l') \cup \{pkt'\})][l \mapsto (\mathtt{Q}(l) \setminus \{pkt\})] \end{array}}{(\mathtt{Q},\Delta) \xrightarrow{(l,pkt)/(l',pkt')}_{NF} (\mathtt{Q}',\Delta')}$$

NET-LINK

$$\frac{\begin{array}{c} topo(l) = l' \qquad pkt \in \mathtt{Q}(l) \\ \mathtt{Q}' = \mathtt{Q}[l' \mapsto (\mathtt{Q}(l') \cup \{pkt\})][l \mapsto (\mathtt{Q}(l) \setminus \{pkt\})] \end{array}}{(\mathtt{Q},\Delta) \xrightarrow{(l,pkt)/(l',pkt')}_{null} (\mathtt{Q}',\Delta)}$$

NET-TRANS-IN

$$\frac{l \in \mathsf{IngressLocs} \qquad \mathtt{Q}' = \mathtt{Q}[l \mapsto (\mathtt{Q}(l) \cup \{pkt\})]}{(\mathtt{Q},\Delta) \xrightarrow{-/(l,pkt)}_{null} (\mathtt{Q}',\Delta)}$$

Figure 12: Semantics of packet interleaving execution.

We call a network N *processing-order preserving* if there is a packet order $\prec$ such that for all $E \in \mathsf{E}^\infty(N)$, E is order preserving under $\prec$. Some example processing-order preserving networks includes: (1) a network with no stateful NFs; (2) a network where any single packet only traverses one stateful NF; (3) a network with connection-based NFs where packets in a connection are delivered in order.

Equivalence between one packet model and packet interleaving model: Given a closed network execution trace $E = s_0 \xrightarrow{lp_1/lp_2} \cdots \xrightarrow{lp_{n-1}/lp_n} s_n$ in $\mathsf{E}^\infty(N)$ of a network N and an ID id, we call the per-packet trace of id, denoted $E|_{id}$, as the sequence $[lp_{i_1}/lp_{i_1+1} \ldots lp_{i_k}/lp_{i_k+1}]$ obtained by projecting all lp_i/lp_{i+1} pairs (except for the first pair corresponding to the NET-TRANS-IN rule) with the ID id from the sequence $[lp_1/lp_2 \ldots lp_{n-1}/lp_n]$. We define the per-packet trace for executions in the one-packet model similarly.

Lemma 1 *Given an order-preserving network N, $\forall E \in \mathsf{E}^\infty(N)$, $\exists E' \in \mathsf{E}^{\mathsf{one}}(N)$, s.t. $\forall id$, $E|_{id} = E'|_{id}$.*

PROOF. Let $\prec$ be the packet order satisfying the order-preserving of N. Suppose $E = s_0 \xrightarrow{lp_0/lp'_0} s_1 \xrightarrow{lp_1/lp'_1}, \ldots, \xrightarrow{lp_{n-1}/lp'_{n-1}} s_n$. We define the out-of-order index I of E as number of disordered transitions w.r.t. $\prec$. Formally $I(E,\prec) = \sum_{i<n} |\{j | j < i, lp_i \prec lp_j\}|$.

We prove this lemma by induction over the out-of-order index of E.

Base case: Since $I(E,\prec) = 0$, we know that $lp_i \not\prec lp_{i-1}$ for all

$i \geq 1$, i.e., $ID(lp_{i-1}) = ID(lp_i)$ or $ID(lp_{i-1}) \prec ID(lp_i)$. Since E is closed, there is an execution trace $(lp_0, \Delta_0) \to (lp_1, \Delta_1) \to \ldots \to (lp_n, \Delta_n)$ (if some lp_i is empty, simply ignore that state) in the one-packet model where Δ_i is the table valuation in s_i, and the per-packet trace for all packets are the same.

Inductive case: We have the inductive hypothesis: For all $E \in \mathsf{E}^\infty(N)$ where $I(E, \prec) \leq k$, there is an execution trace $E' \in \mathsf{E}^{\mathsf{one}}(N)$ s.t. $E|_{id} = E'|_{id}$. Now consider the case where $I(E, \prec) = k+1$. Since $I(E, \prec) > 0$, there exists $i > 0$ such that $s_{i-1} \xRightarrow{lp_{i-1}/lp'_{i-1}} s_i \xRightarrow{lp_i/lp'_i} s_{i+1}$ and $lp_i \prec lp_{i-1}$. We claim that there must exists a network state s'_i such that $s_{i-1} \xRightarrow{lp_i/lp'_i} s'_i \xRightarrow{lp_{i-1}/lp'_{i-1}} s_{i+1}$. This is easy to see when the transition $s_i \xRightarrow{lp_i/lp'_i} s_{i+1}$ is obtained from rule NET-LINK or NET-TRANS-IN. Suppose the transition is from NET-TRANS-NF and the NF that processes lp_i on s_i is NF. If transition $s_{i-1} \xRightarrow{lp_{i-1}/lp'_{i-1}} s_i$ is not from NET-TRANS-NF or does not correspond to the processing NF NF, the claim is also obvious. When both transitions correspond to the process of the packets on NF, from the processing-order preserving definition, lp_i must be non-conflicting with lp_{i-1} at NF. Thus from Lemma 6 the claim is still true. By swapping the processing of lp_{i-1} and lp_i we obtain an order-preserving execution trace E'' from E such that $E''|_{id} = E|_{id}$ for all id and $I(E'', \prec) = k$. From the inductive hypothesis, there is an execution trace $E' \in \mathsf{E}^{\mathsf{one}}(N)$ s.t. $E'|_{id} = E''|_{id} = E|_{id}$ for all id. □

Lemma 6 *For all network N, located packets lp_1, lp'_1, lp_2, lp'_2, $NF \in N$, and network states s_1, s_2, s_3 of N, if $s_1 \xRightarrow{lp_1/lp'_1}_{NF} s_2 \xRightarrow{lp_2/lp'_2}_{NF} s_3$ and lp_1 is non-conflicting with lp_2 at NF, then $\exists s'_2$ s.t. $s_1 \xRightarrow{lp_2/lp_2}_{NF} s'_2 \xRightarrow{lp_1/lp'_1}_{NF} s_3$.*

PROOF. Immediate from definition of non-conflicting.

D Soundness and Completeness of Checking Per-Packet-Trace Policies

From Lemma 1, we can show that checking a range of policies involving per-packet traces is equivalent between the packet interleaving network model and the one packet model provided that network is processing-order preserving.

Per-packet-trace policies. We define per-packet-trace policies (for the one-packet model) as follows.

$$\varphi^{one} \ ::= \forall \overrightarrow{x_i \in D_i}.\mathtt{G}\,(\gamma_1 \to \mathtt{g}\gamma_2) \,|\, \forall \overrightarrow{x_i \in D_i}.\mathtt{G}\,(\gamma_1 \to \gamma_2)$$

A translation function, denoted $\langle \cdot \rangle$, turns a formula for the one-packet model to a corresponding formula for the packet-interleaving model. It is defined as follows, which essentially introduces a packet ID into the formula. We only show selected rules and the rest are inductively defined over the structure of the formula.

$$\begin{aligned}
&\langle \forall \overrightarrow{x_i \in D_i}.\mathtt{G}\,(\gamma_1 \to \mathtt{g}\gamma_2) \rangle = \\
&\quad \forall id. \forall \overrightarrow{x_i \in D_i}.\mathtt{G}\,(\langle \gamma_1 \rangle_{id} \to \\
&\qquad \langle \gamma_2 \rangle_{id}\, \mathtt{U}\,(id.\mathtt{loc} = \mathsf{Drop} \vee id.\mathtt{loc} = \mathsf{Exit})) \\
&\langle f = x \rangle_{id} = id.f = x \\
&\langle \mathtt{loc} = x \rangle_{id} = id.\mathtt{loc} = x \\
&\langle f \rangle_{id} = id.f \quad \langle x \rangle_{id} = x \quad \langle v \rangle_{id} = v
\end{aligned}$$

We can prove the following theorem.

Theorem 7 *For all order-preserving network N, $N \models^{one} \varphi^{one}$ if and only if $N \models^\infty \langle \varphi^{one} \rangle$.*

PROOF. We only prove the case for the first form of φ^{one}. Proofs for the second is very similar. Let $\varphi^{one} = \forall \overrightarrow{x_i \in D_i}.\mathtt{G}\,(\gamma_1 \to \mathtt{g}\gamma_2)$.

($\leftarrow$) Suppose $N \not\models^{one} \varphi^{one}$. By the definition of the policy, there exists $\overrightarrow{v_i}$ for $\overrightarrow{x_i}$ and an execution trace $E = (lp_0, \Delta_0) \to \ldots \to (lp_n, \Delta_n) \in \mathsf{E}^{\mathsf{one}}(N)$, and some i, j, such that $0 \leq i \leq j \leq n$, $(V, E[i..]) \models \gamma_1$, and $(V, E[j..]) \not\models \gamma_2$, where $V = [\overrightarrow{x_i \mapsto v_i}]$. and for all k, l_k s.t. $i < k < j$ and $lp_k = (l_k, \_)$, $l_k \notin \{\mathsf{Drop}, \mathsf{Exit}\}$.

By the semantics of one-packet model, $\forall m$ s.t. $i < m < j$, $ID(lp_m) = ID(lp_i) = ID(lp_j) = pid$.

We can then construct an execution trace $E' \in \mathsf{E}^\infty(N)$ by injecting one packet at a time to simulate E, and the only trivial difference between E and E' is that incoming packets takes an extra step, rather than being enqueued right after the previous packet exits.

It's straightforward to show that $(V[id \mapsto pid], E'[i..]) \models \langle \gamma_1 \rangle_{id}$ and $(V[id \mapsto pid], E'[j..]) \not\models \langle \gamma_2 \rangle_{id}$, and $i < k < j$ and $(V[id \mapsto pid], E'[k..]) \models id.\mathtt{loc} \neq \mathsf{Drop} \wedge id.\mathtt{loc} \neq \mathsf{Exit}$, since there is only one packet with ID pid in the packet queue.

By the formula semantics, $(V[id \mapsto pid], E'[i..]) \not\models \langle \mathtt{g}\,\gamma_2 \rangle_{id}$. It follows that $(\emptyset, E') \not\models \langle \varphi^{one} \rangle$, so $N \not\models^\infty \langle \varphi^{one} \rangle$, which contradicts with our assumption.

($\rightarrow$) Suppose $N \not\models^\infty \langle \varphi^{one} \rangle$. By the definition of the policy, there exists $\overrightarrow{v_i}$ for $\overrightarrow{x_i}$, pid for id and an execution trace $E = s_0 \Rightarrow \ldots \Rightarrow s_n \in \mathsf{E}^\infty(N)$, and some i, j, such that $0 \leq i \leq j \leq n$, $(V, E[i..]) \models \langle \gamma_1 \rangle_{id}$, and $(V, E[j..]) \not\models \langle \gamma_2 \rangle_{id}$, where $V = [\overrightarrow{x_i \mapsto v_i}, id \mapsto pid]$. And for all k s.t. $i < k < j$, $(V, E[k..]) \models id.\mathtt{loc} \neq \mathsf{Drop} \wedge id.\mathtt{loc} \neq \mathsf{Exit}$.

By the network semantics, the located packet with ID pid in E_i must have entered the queue at some point. Let $i' \leq i$ be the index of the first such state in E, such that $E = \cdots \xRightarrow{lp_{i'}/lp'_{i'}} s_{i'} \cdots$. Similarly, we identify $j' \leq j$ such that j' is the first state where the located packet in E_j has arrived at the packet queue. W.o.l.g. $E = \cdots \xRightarrow{lp_{i'}/lp'_{i'}} s_{i'} \cdots \xRightarrow{lp_{j'}/lp'_{j'}} s_{j'} \cdots$. It is straightforward that for all k s.t. $i' < k < j'$, $(V, E[k..]) \models id.\mathtt{loc} \neq \mathsf{Drop} \wedge id.\mathtt{loc} \neq \mathsf{Exit}$ (Dropped or Exited packets cannot come back with the same pid). Note that $ID(lp'_{i'}) = ID(lp'_{j'}) = pid$.

By Lemma 1, there is an execution trace $E' \in \mathsf{E}^{\mathsf{one}}(N)$ where $E'|_{pid} = E|_{pid}$. By the semantics of the one packet model, $E' = \cdots E'' \cdots$, where $E''|_{pid} = E|_{pid}$ and E'' contains processing of only packets with ID pid. So, $(V\backslash id, E') \not\models \gamma_1 \rightarrow \mathtt{g}\,\gamma_2$.

It follows that $N \not\models^{one} \varphi^{one}$, which contradicts with our assumption. □

Corollary 8 *NetSMC is sound and complete w.r.t. the packet interleaving model when checking isolation, tag preservation, tag-based isolation policies.*

This follows from Theorem 7 and Theorem 3.

Comparison with VMN: VMN accepts policies of the form: $\forall n, p : G\neg(rcv(d,n,p) \wedge predicate(p))$ (see Appendix B.2 in [41]) where $predicate(p)$ may include past events. We can express all the policies in VMN that do not have past events in $predicate(p)$ (i.e., a basic formula). Isolation, tag-based isolation, and tag preservation fall into this category. Further, in the three cases. NetSMC is sound and complete w.r.t. VMN.

The conditional isolation in VMN cannot be expressed in our policy language, since it involves past events and requires a generic until operator, which we do not support. On the other hand, NAT consistency, conditional reachability, flow affinity, double NAT, and dynamic service chaining cannot be expressed in VMN. In these cases, NetSMC is sound and conditional complete w.r.t. the one-packet model.

E Policy Translation to CTL

Our choice of the policy language as a subset of LTL allows us to translate policies to equivalent forms in CTL. Thus we can use the model checking algorithm of CTL to check those policies. To simplify our notation in the proof, we write $(V,s) \models \rho$ to denote that $\forall E$ s.t. $E_0 = s$, $(V,E) \models \rho$.

Theorem 9 *For all network N and policy P, $N \models P$ if and only if $N \models P_{CTL}$.*

PROOF. Suppose $P = \forall \overrightarrow{x_i \in D_i}.\rho$ and thus $P_{CTL} = \forall \overrightarrow{x_i \in D_i}.\rho_{CTL}$. (If) Since $N \models P_{CTL}$, by the definition over the structure of ρ $\forall V, s$ s.t. $V[x_i] \in D_i$ and s is an initial state of N, $(V,s) \models \rho_{CTL}$. By Lemma 10, $(V,s) \models \rho$. Thus $N \models P$. (Only if) Since $N \models P$, by the definition, $\forall V, s$ s.t. $V[x_i] \in D_i$ and s is an initial state of N, $(V,s) \models \rho$. By Lemma 10, $(V,s) \models \rho_{CTL}$. Thus $N \models P_{CTL}$. □

Lemma 10 *For all network N, temporal formula ρ, valuation function V for variables appeared in ρ, state s in N, $(V,s) \models \rho$ if and only if $(V,s) \models \rho_{CTL}$.*

PROOF. Proof by induction over the structure of ρ.
Case 1 $\rho = \gamma$: This is immediate from the definition.
Case 2 $\rho = \mathtt{F}\gamma$: (If) Since $(V,s) \models \mathtt{AF}\gamma$, for all execution trace E where $E_0 = s$, there is some $i \geq 0$ s.t. $E_i \models \gamma$. Thus, $(V,s) \models \mathtt{F}\gamma$. (Only if) Since $(V,s) \models \mathtt{F}\gamma$, for all execution trace E where $E_0 = s$, there is some $i \geq 0$ s.t. $E_i \models \gamma$. Thus, $(V,s) \models \mathtt{AF}\gamma$.
Case 3 $\rho = \mathtt{f}\gamma$: (If) Since $(V,s) \models \mathtt{A}((\mathtt{loc} \neq \mathsf{Drop} \wedge \mathtt{loc} \neq \mathsf{Exit})\mathtt{U}\gamma)$, for all execution trace E where $E_0 = s$, there is some $i \geq 0$ s.t. $E_i \models \gamma$ and for all $j < i$, the location of E_j is not Drop nor Exit. Thus, $(V,E) \models \mathtt{f}\gamma$. Thus, $(V,s) \models \mathtt{f}\gamma$. (Only if) Since $(V,s) \models \mathtt{f}\gamma$, for all execution trace E where $E_0 = s$, there is some $i \geq 0$ s.t. $E_i \models \gamma$ and for all $j < i$ the location of E_j is not Drop nor Exit. Thus, $(V,s) \models \mathtt{A}((\mathtt{loc} \neq \mathsf{Drop} \wedge \mathtt{loc} \neq \mathsf{Exit})\mathtt{U}\gamma)$.
Case 4 $\rho = \mathtt{G}(\gamma \rightarrow \rho')$: From Lemma 12, we only need to show for all s', $(V,s') \models \gamma \rightarrow \rho'$ if and only if $(V,s') \models \gamma \rightarrow \rho_{CTL}$. First, when $(V,s') \not\models \gamma$, we have $(V,s') \models \gamma \rightarrow \rho'$ and $(V,s') \models \gamma \rightarrow \rho_{CTL}$. Second, when $(V,s') \models \gamma$, from the inductive hypothesis, $(V,s') \models \rho'$ if and only if $(V,s') \models \rho_{CTL}$.
Case 5 $\rho = \mathtt{g}(\gamma \rightarrow \rho_1)$: Similar to the proof of Case 4 and use Lemma 11.
Case 6 $\rho = \mathtt{X}(\gamma \rightarrow \rho_1)$: Similar to the proof of Case 4 and use Lemma 12. □

Note that the ρ in the following two lemmas are generic temporal formulas, not confined to our policy syntax.

Lemma 11 *For all network N, temporal formula ρ, valuation function V for variables appeared in ρ, if for all state s, $(V,s) \models \rho$ is equivalent to $(V,s) \models \rho_{CTL}$, then for all state s', $(V,s') \models \mathtt{g}\rho$ is equivalent to $(V,s') \models \mathtt{A}((\rho_{CTL})\mathtt{U}(\mathit{loc} = \mathsf{Drop} \vee \mathit{loc} = \mathsf{Exit}))$.*

PROOF. (If) By the definition of $(V,s') \models \mathtt{A}((\rho_{CTL})\mathtt{U}(\mathsf{Drop} \vee \mathsf{Exit}))$, for all execution trace E where $E_0 = s'$ and $i \geq 0$ where the location of E_i is Drop or Exit, $(V,E_j) \models \rho_{CTL}$ for all $j < i$. By the assumption, $(V,E_j) \models \rho$. Thus, $(V,E[j..]) \models \rho$. Therefore, $(V,E) \models \mathtt{g}\rho$. Thus, $(V,s') \models \mathtt{g}\rho$.
(Only if) Suppose $(V,s') \not\models \mathtt{A}((\rho_{CTL})\mathtt{U}(\mathtt{loc} = \mathsf{Drop} \vee \mathtt{loc} = \mathsf{Exit}))$. By its definition, there is a execution trace E and $i \geq 0$ such that $E_0 = s'$ and $(V,E_i) \not\models \rho_{CTL}$ and for all $j \leq i$, the location of E_j is not Drop nor Exit. Since $(V,E_i) \not\models \rho_{CTL}$, we have $(V,E_i) \not\models \rho$; i.e. there is a execution trace E' where $E'_0 = E_i$ such that $(V,E') \not\models \rho$. Consider the execution trace $E'' = E[0..i-1] + +E'$. Note that $E''[i..] = E'$, thus $(V,E''[i..]) \not\models \rho$. In addition, for all $j \leq i$, the location of E''_j is not Drop nor Exit. Thus $(V,E'') \not\models \mathtt{g}\rho$, which contradicts to that $(V,s') \models \mathtt{g}\rho$. □

Lemma 12 *For all network N, temporal formula ρ, valuation function V for variables appeared in ρ, if for all state s, $(V,s) \models \rho$ is equivalent to $(V,s) \models \rho_{CTL}$, then for all state s', $(V,s') \models \mathtt{G}\rho$ is equivalent to $(V,s') \models \mathtt{AG}(\rho_{CTL})$ and $(V,s') \models \mathtt{X}\rho$ is equivalent to $(V,s') \models \mathtt{AX}(\rho_{CTL})$.*

PROOF. Similar to above. □

F Formal Specification of Example Policies

Isolation: Packets sent from host A can never reach host B:

$$\mathtt{G}(\mathtt{loc} = A \rightarrow \mathtt{g}(\mathtt{loc} \neq B))$$

Tag-based isolation: Packets tagged with T cannot reach the middlebox MB.

$$\mathtt{G}(\mathtt{tag} = T \rightarrow \mathtt{g}(\mathtt{loc} \neq MB))$$

Tag preservation: The tag T should not be modified by NFs.

$$\mathtt{G}(\mathtt{tag} = T \rightarrow \mathtt{g}(\mathtt{tag} = T))$$

NAT consistency: If a NAT modifies a packet's port then all future packets in the flow should have the same port.

$$\begin{aligned} \forall i.\quad & \mathtt{G}(\mathtt{flow} = i \wedge \mathtt{loc} = NAT\_IN \rightarrow \\ & \quad \mathtt{X}(\mathtt{srcport} = p \wedge \mathtt{loc} = NAT\_OUT \rightarrow \\ & \quad\quad \mathtt{G}(\mathtt{flow} = i \wedge \mathtt{loc} = NAT\_IN \rightarrow \\ & \quad\quad\quad \mathtt{X}(\mathtt{srcport} = p)))) \end{aligned}$$

Conditional reachability: Whenever a packet sent from a host A reaches host B, all packets sent from host B afterwards can reach A.

$$\mathtt{G}(\mathtt{loc} = A \rightarrow g(\mathtt{loc} = B \rightarrow \mathtt{G}(\mathtt{loc} = B \rightarrow f(\mathtt{loc} = A))))$$

Flow affinity: Packets in the same flow should be load-balanced ot the same server.

$$\begin{aligned} \forall i.\quad & \mathtt{G}(\mathtt{flow} = i \wedge \mathtt{loc} = S_1 \rightarrow \\ & \quad \mathtt{G}(\mathtt{loc} = C \wedge \mathtt{flow} = i \rightarrow \\ & \quad\quad \mathtt{f}(\mathtt{loc} = S_1))) \end{aligned}$$

Double NAT: Packets from the outside can always reach the inside (despite two NATs).

$$\mathtt{G}(\mathtt{loc} = OUT \rightarrow \mathtt{f}(\mathtt{loc} = IN))$$

Dynamic service chaining: After a host from *Dept* has sent more than 10 suspicious packets, all of its packets should pass heavy IPS H.

$$\begin{aligned} \forall x \in Dept.\quad & \mathtt{G}(\mathtt{src} = x \wedge susp[x] > 10 \rightarrow \\ & \quad \mathtt{G}(\mathtt{src} = x \rightarrow \mathtt{f}(\mathtt{loc} = H))) \end{aligned}$$

Tiramisu: Fast Multilayer Network Verification

Anubhavnidhi Abhashkumar*, Aaron Gember-Jacobson†, Aditya Akella*
University of Wisconsin - Madison, Colgate University†*

Abstract: Today's distributed network control planes are highly sophisticated, with multiple interacting protocols operating at layers 2 and 3. The complexity makes network configurations highly complex and bug-prone. State-of-the-art tools that check if control plane bugs can lead to violations of key properties are either too slow, or do not model common network features. We develop a new, general multilayer graph control plane model that enables using fast, property-customized verification algorithms. Our tool, Tiramisu can verify if policies hold under failures for various real-world and synthetic configurations in $<$ 0.08s in small networks and $<$ 2.2s in large networks. Tiramisu is 2-600X faster than state-of-the-art without losing generality.

1 Introduction

Many networks employ complex topologies and distributed control planes to realize sophisticated network objectives. At the topology level, networks employ techniques to virtualize multiple links into logically isolated broadcast domains (e.g., VLANs) [8]. Control planes employ a variety of routing protocols (e.g., OSPF, eBGP, iBGP) which are configured to exchange routing information with each other in intricate ways [9, 21]. Techniques to virtualize the control plane (e.g., virtual routing and forwarding (VRF)) are also common [8].

Bugs can easily creep into such networks through errors in the detailed configurations that the protocols need [9, 21]. In many cases, bugs are triggered when a failure causes the control plane to reconverge to new paths. Such bugs can lead to a variety of catastrophic outcomes: the network may suffer from reachability blackholes [5]; services with restricted access may be rendered wide open [4]; and, expensive paths may be selected over inexpensive highly-preferred ones [4].

A variety of tools attempt to verify if networks could violate important policies. In particular, *control plane analyzers* [6, 12–14, 23, 29] proactively verify if the network satisfies policies against various environments, e.g., potential failures or external advertisements. State-of-the-art examples include: graph-algorithm based tools, such as ARC [14] which models all paths that may manifest in a network as a series of weighted digraphs; satisfiability modulo theory (SMT) based tools, such as Minesweeper [6] which models control planes by encoding routing information exchange, route selection, and failures using logical constraints/variables; and, explicit-state model checking (ESMC) based tools, such as Plankton [23][1] which models routing protocols in a custom language, such that an explicit state model checker can explore the many possible data plane states resulting from the control plane's execution.

Unfortunately, these modern control plane tools still fall short because they make a hard trade-off between performance and generality (§2). ARC abstracts many low level control plane details which allows it to leverage polynomial time graph algorithms for verification, offering the best performance of all tools. But the abstraction ignores many network design constructs, including commonly-used BGP attributes, and iBGP. While these are accounted for by the other classes of tools [6, 23] that model control plane behavior at a much lower level, the tools' detailed encoding renders verification performance extremely poor, especially when exploring failures (§8). Finally, all existing tools ignore VLANs, and VRFs.

This paper seeks a fast general control plane verification tool that also accounts for layer 2.5 protocols, like VLANs.

We note that today's trade-off between performance and generality is somewhat artificial, and arises from an unnatural coupling between the control plane encoding and the verification algorithm used. For example, in existing graph-based tools, graph algorithms are used to verify the weighted digraph control plane model. In SMT-based tools, the detailed constraint-based control plane encoding requires a general constraint solver to be used to verify any policy. ESMC-based tools' encoding forces a search over the many possible data plane states, mirroring software verification techniques that exhaustively explore the execution paths of a general program.

The key insight in our framework, Tiramisu, is to decouple encoding from verification algorithms. Tiramisu leverages a *new, rich encoding* for the network that models various control plane features and network design constructs. The encoding allows Tiramisu to use different *custom verification algorithms* for different *categories of policies* that substantially improve performance over the state-of-the-art.

Tiramisu's network model uses graphs as the basis, similar to ARC. However, the graph model is multi-layered and uses multi-attribute edge weights, thereby capturing dependencies among protocols (e.g., iBGP depending on OSPF-provided paths) and among virtual and physical links, and accounting for filters, tags, and protocol costs/preferences.

For custom verification, Tiramisu notes that most policies studied in the literature can be grouped into three categories (Table 1): (i) policies that require the actual path that manifests in the network under a given failure; (ii) policies that care about certain quantitative properties of paths that may manifest (e.g., maximum path length); and, finally, (iii) policies that merely care about whether a path exists. Tiramisu leverages the underlying model's graph structure to develop performant verification algorithms for each category.

To verify category (i) policies, Tiramisu efficiently solves the stable paths problem [15] using the *Tiramisu Path Vector Protocol* (*TPVP*). *TPVP* simulates control plane computa-

[1]Plankton was developed contemporaneously with our system

tions across multiple devices and interdependent protocols by operating on the Tiramisu network model. *TPVP*'s domain-specific path computation is faster than the general search strategies used in SMT solvers, making Tiramisu orders of magnitude faster in verifying category (i) policies. *TPVP* also outperforms ESMC-based tools, because *TPVP*'s use of routing algebra [16, 25] atop the Tiramisu rich graph allows it to compute paths in one shot, whereas ESMC-based tools emulate protocols, and explore their state, one at a time in order to account for inter-protocol dependencies.

For category (ii), Tiramisu's insight is to use integer linear program (ILP) formulations that only model the variables that are relevant to the policy being verified. Tiramisu significantly outperforms SMT- and ESMC-based tools, whose computation of specific paths requires them to (needlessly) explore a much larger variable search space.

For category (iii), Tiramisu uses a novel graph traversal algorithm to check path existence. The algorithm invokes canonical depth-first search on multiple Tiramisu subgraphs, to account for tags (e.g., BGP communities) whose use on a path controls the path's existence. For such policies, Tiramisu matches ARC's performance for simple control planes; but Tiramisu is much more general.

Finally, for many of the policies in categories (i) and (ii), Tiramisu leverages the underlying model's graph structure to develop a graph algorithm-based accelerator for further verification speed-up. We show how to use a variant of a dynamic programming-based algorithm for the k shortest paths problem [31] to curtail needlessly exploring paths that manifest under many not-so-relevant failure scenarios.

We implemented Tiramisu in Java [2] and evaluated it with many real and synthetic configurations, spanning campus, ISP, and data center networks. We find that Tiramisu significantly outperforms the state-of-the-art, and is more general—some of the networks have layer-2/3 features that Minesweeper and Plankton don't model. Using category-specific algorithms on complex networks, Tiramisu verified category i, ii, and iii policies in *60*, *80* and *3ms*, respectively. Compared to Minesweeper, Tiramisu is *80X*, *50X*, and *600X* faster for category i, ii, and iii policies, respectively. On iBGP networks, Tiramisu outperforms Plankton by up to *300X* under failures. Tiramisu's *TYEN* acceleration improves performance by *1.3-3.8X*. Tiramisu scales well, providing verification results in $\sim$100ms per traffic class for networks with $\sim$160 routers.

2 Motivation

In this section, we provide an overview of state-of-the-art control plane verifiers. We then identify their key drawbacks that motivate Tiramisu's design.

2.1 Existing control plane verifiers

State-of-the-art control plane verifiers are based on three different techniques: graph algorithms [14], symbolic model checking [6,12,29], and explicit-state model checking [13,23]. We review the most advanced verifier in each category.

Graph algorithms: ARC [14] models a network's control plane using a set of directed graphs. Each graph encodes the network's forwarding behavior for a specific traffic class–i.e., packets with specific source and destination subnets. In the graphs, vertices represent routing processes (i.e. instances of routing protocols running on specific devices); directed edges represent possible network hops enabled by the exchange of advertisements between processes; edge weights encode OSPF costs or AS path lengths. ARC verifies a policy by checking a simple graph property: e.g, *src* and *dst* are always blocked if they are in separate components. By leveraging graph algorithms, ARC offers orders-of-magnitude better performance [14] than simulation-based verifiers [13]. However, ARC's simple graph model does not cover: widely used layer-3 protocols (e.g., iBGP), any layer-2 primitives (e.g., VLANs), and many protocol attributes (e.g., BGP community).

Symbolic model checking: Minesweeper [6] verifies a network's policy compliance by formulating and solving a satisfiability modulo theory (SMT) problem. The SMT constraints encode the behavior of the network's control and data planes (M) as well as the negation of a target policy ($\neg P$). If the SMT constraints ($M \wedge \neg P$) are unsatisfiable, then the policy is satisfied. To provide extensive network design coverage, Minesweeper uses many variables, which results in a large search space. Minesweeper verifies all policies on this large general SMT model. To verify for k failures, Minesweeper's SMT solver may, in the worst case, enumerate all possible combinations of k-link failures.

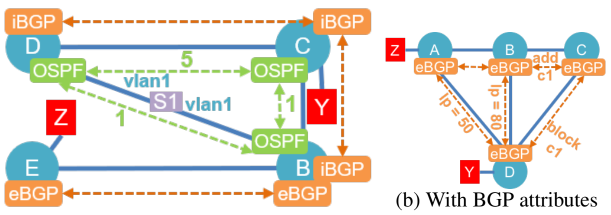
Explicit-state model checking: Plankton [23] models different control plane protocols (e.g., OSPF, BGP) using the Path Vector Protocol (PVP) [15] in a modeling language (Promela). It tracks dependencies among protocols defined in a network's control plane based on packet equivalence classes (PECs). It then uses an explicit state model checker, SPIN, to search on the overall state space of this model and find a state that violates a policy. To speedup verification, Plankton runs multiple independent SPIN instances in parallel. Plankton uses other optimizations including device equivalence to reduce the failure scenarios to explore. Despite these optimizations, Plankton still enumerates the state space of many failure scenarios, and checks them sequentially (§8).

2.2 Challenges

We now identify three high-level challenges that affect existing verifiers' performance and/or correctness.

Cross-layer dependencies. Consider the network in Figure 1a. Router B is an eBGP peer of router E, and router C is an iBGP peer of routers B and D. B and D are connected to switch $S1$ on VLAN 1. All routers except E belong to the same OSPF domain; the cost of link C–D is 5 and others is 1.

In this network, C learns a route to E through its iBGP neighbor B; the route's next hop is the IP address of B's loopback interface. In order for C to route traffic through B,



(a) With protocol dependencies (b) With BGP attributes

Figure 1: Example networks

C must compute a route to *B* using OSPF. The computed path depends on the network's failure state. In the absence of failures, *OSPF* prefers path $C \rightarrow B$ (cost 1). When the link *B–C* fails, *OSPF* prefers a different path: $C \rightarrow D \rightarrow B$ (cost 6). Unfortunately, traffic for *E* is dropped at *D*, because *D* never learns a route to *E*; *E* is not in the same OSPF domain, and routes learned (by *C*) via iBGP are not forwarded. If *B* and *D* were iBGP peers, or *B* redistributed its *eBGP*-learned routes to *OSPF*, then this blackhole would be avoided.

ARC's simplistic graph abstraction cannot model iBGP and thus cannot model iBGP-OSPF dependencies. Hence, ARC cannot be used to verify policies in this network. Minesweeper can model iBGP, but its encoding is inefficient. To model iBGP, Minesweeper creates *n* additional copies of the network where *n* represents number of routers running iBGP. Each copy models forwarding towards the next-hop address associated with each iBGP router. This increases Minesweeper's SMT model size by *n*X, which significantly degrades its performance. In Plankton, the iBGP-OSPF dependency is encoded as a dependency between PECs, which prevents Plankton from fully parallelizing its SPIN instances. Hence, Plankton loses the performance benefits of parallelism.

Cross-layer dependencies also impact other network scenarios. Assume the $B - S1$ link in Figure 1a was assigned to VLAN 2. Now *B* and *D* are connected to the same switch $S1$ on different VLANs; internally, $S1$ runs two *virtual switches*, one for each VLAN. Hence, traffic between *B* and *D* cannot flow through switch $S1$. By default, ARC, Minesweeper, and Plankton, assume layer-2 connectivity. Thus, according to these verifiers, *B* and *D* are reachable and traffic can flow between them, which is incorrect.

The overall theme is that protocols "depend" on each other—e.g., iBGP depends on OSPF, BGP and OSPF depend on VLANs, etc.—and these dependencies must be *fully*, *correctly* and *efficiently* modeled.

Protocol attributes. Consider the network in Figure 1b. All routers (*A–D*) run eBGP. *B* adds community "c1" to the advertisements it sends to *C*, and *D* blocks all advertisements from *C* with community "c1". Additionally, *D* prefers routes learned from *B* over *A* by assigning local preference (lp) values 80 and 50, respectively.

The path that *D* uses to reach *A* depends on communities and local preference. There are three physical paths from *D* to *A*: (*i*) $D \rightarrow A$, (*ii*) $D \rightarrow B \rightarrow A$, and, (*iii*) $D \rightarrow C \rightarrow B \rightarrow A$. However, since router *B* adds community "c1" to routes advertised to *C*, and *D* blocks advertisements from *C* with this community, path *iii* is unusable. Furthermore, path *ii* is preferred over the shorter path *i* due to local preference.

Since ARC uses Dijkstra's algorithm to compute paths, it can only model additive path metrics like OSPF cost and AS-path length; it cannot model non-additive properties such as local preference, communities, etc. Hence, ARC incorrectly concludes that path *iii* is valid and (shortest) path *i* is preferred. Although Minesweeper and Plankton can model these attributes, they suffer from other drawbacks mentioned earlier.

Failures. Assume there are no communities in the network in Figure 1b, and routers *A* and *D* are configured to run OSPF in addition to eBGP. This network can tolerate a single link failure without losing connectivity.

According to ARC, traffic from *D* to *A* can flow through four paths: $D_{bgp} \rightarrow C_{bgp} \rightarrow B_{bgp} \rightarrow A_{bgp}$, $D_{bgp} \rightarrow B_{bgp} \rightarrow A_{bgp}$, $D_{bgp} \rightarrow A_{bgp}$, and $D_{ospf} \rightarrow A_{ospf}$. To evaluate the network's failure resilience, ARC calculates the *min-cut* of this graph, which is 3, and concludes that it can withstand two arbitrary, simultaneous link failures. This is incorrect because edges $D_{ospf} \rightarrow A_{ospf}$ and $D_{bgp} \rightarrow A_{bgp}$ are both unusable when the physical link *D–A* fails.

As mentioned in §2.1, Minesweeper and Plankton enumerate multiple failure scenarios and this makes them very slow to verify for failures. For example, in this 5-link network, to *verify reachability with 1 failure*, Minesweeper (and Plankton without optimization) may explore 5 failure scenarios before establishing reachability.

An overall issue is that irrespective of the policy, Minesweeper and Plankton need to compute the actual path taken in the network. ARC, on the other hand, represents policies as graph properties, e.g. mincut, connectivity, etc. It then uses fast polynomial time algorithms to compute these properties. However, ARC's simplistic graph abstraction cannot model all network features. Our goal is to create a tool that combines the network coverage of Minesweeper and Plankton, with the performance benefits of ARC.

3 Overview

Motivated by the inefficiencies and coverage limitations of existing network verifiers (§2), we introduce a new network verification tool called Tiramisu.Tiramisu is rooted in a rich graph-based network model that captures forwarding and failure dependencies across multiple routing and switching layers (e.g., BGP, OSPF, and VLANs). Since routing protocols are generally designed to operate on graphs and their constituent paths, graphs offer a natural way to express the route propagation, filtering, and selection behaviors encoded in device configurations and control logic. Moreover, graphs admit efficient analyses that allow Tiramisu to quickly reason about important network policies—including reachability, path lengths, and path preferences—in the context of multi-link failures. In this section, we highlight how Tiramisu's graph-based model and verification algorithms address the challenges discussed in §2. Detailed descriptions, algorithms, and proofs of cor-

Category	Policy	Meaning	Comments
i: compute path with *TPVP*	PREF	Path Preference	Can use *TYEN*
	MC	Multipath Consistency	
ii: compute actual numeric graph property with ILP	KFAIL	Reachability < K failures	Can use *TYEN*
	BOUND	All paths have length < K	
	EB	All paths have equal length	
iii: identify connectivity wth *TDFS*	BLOCK	Always Blocked	
	WAYPT	Always Waypointing	
	CW	Always Chain of Waypoints	
	BH	No Blackhole	

Table 1: Policies Verified

rectness, are presented in later sections.

3.1 Graph-based network model

To accurately and efficiently model cross-layer dependencies, Tiramisu constructs two inter-related types of graphs: *routing adjacencies graphs* (RAGs) and *traffic propagation graphs* (TPGs). The former allows Tiramisu to determine which routing processes may learn routes to specific destinations. The latter models more detail, especially, all the prerequisites for such learning to occur—e.g., OSPF must compute a route to an iBGP neighbor in order for the neighbor to receive BGP updates. Our verification algorithms run on the TPGs.

Routing adjacencies graph. A RAG (e.g, Figure 2a) encodes routing *adjacencies*. Two routing processes are adjacent if they are configured as neighbors (e.g., BGP) or configured to operate on interfaces in the same layer-2 broadcast domain (e.g., OSPF). A RAG contains a vertex for each routing process, and a pair of directed edges for each routing adjacency. Tiramisu runs a domain-specific "tainting" algorithm on the RAG to determine which routing processes may learn routes for a given prefix p.

Traffic propagation graph. In addition to routing adjacencies, propagation of traffic requires: (*i*) a route to the destination, (*ii*) layer-2 and, in the case of BGP, layer-3 routes to adjacent processes, and (*iii*) physical connectivity. A TPG (e.g., Figure 2b) encodes these *dependencies*. Vertices are created for each VLAN on each device and each routing process. Directed edges model the aforementioned dependencies as follows: (*i*) a VLAN vertex is connected to an OSPF/BGP vertex associated with the same router if, according to the RAG, the process may learn a route to a given subnet; (*ii*) an OSPF vertex is connected to the vertices for the VLANs on which it operates, and a BGP vertex is connected to an OSPF and/or VLAN vertex associated with the same router; (*iii*) a VLAN vertex is connected to a vertex for the same VLAN on another device if the devices are physical connected. Additionally, multi-attribute edge labels are assigned to edges to encode filters and "costs" associated with a route. With this structure, Tiramisu is able to correctly model a much wider range of networks than state-of-the-art graph-based models [14]. All verification procedures operate on the TPG.

3.2 Verification algorithms

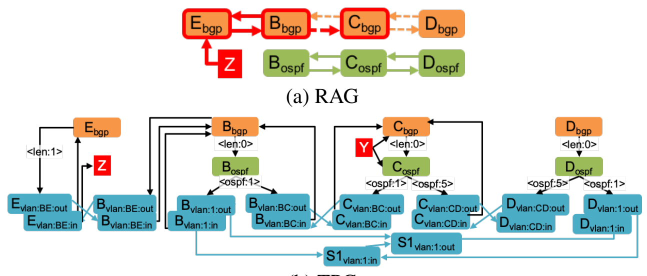
Tiramisu's verification process is rooted in the observation that network policies explored in practice and in academic research (Table 1) fall into three categories: (*i*) policies concerned with the actual path taken under specific failures—e.g., path preference (PREF); (*ii*) policies concerned with quantitative path metrics—e.g., how many paths are present (KFAIL) and bounds on path length (BOUND); and (*iii*) policies concerned with the existence of a path—e.g., blocking (BLOCK) and waypointing (WAYPT). Verifying policies in the first category requires high fidelity modeling of the control plane's output—namely, enumeration/computation of precise forwarding paths—whereas verifying policies in the last category requires low fidelity modeling of the control plane's output—namely, evidence of a single plausible path. For optimal efficiency, Tiramisu's core insight is to introduce category-specific verification algorithms that operate with the minimal level of fidelity required for accurate verification.

TPVP. To verify category *i* policies, Tiramisu efficiently solves the stable paths problem [15] using the *Tiramisu Path Vector Protocol* (*TPVP*). *TPVP* extends Griffin's "simple path vector protocol" (SPVP) [15]. In TPVP, each TPG node consumes the multi-dimensional attributes of outgoing edges, and uses simple arithmetic operations (based on a routing algebra [26]) to select among multiple available paths. In simple networks (that use a single routing protocol) *TPVP* devolves to a distance vector protocol, which computes paths under failures in polynomial time. For such networks, TPVP is comparable to ESMC tools [23], but is faster than general SMT-based tools [6]. For general networks with dependent control plane protocols, TPVP is faster than SMT-based tools, because it uses a domain-specific approach to computing paths, compared to an SMT-based general search strategy. TPVP also beats ESMC tools by emulating the control plane in one shot as opposed to emulating the constituent protocols and exploring their state space one at a time to account for inter-protocol dependencies.

ILP. For category *ii* policies, Tiramisu leverages general integer linear program (ILP) encodings that compute network flows through the TPG, rather than precise paths. The ILPs only consider protocol attributes that impact whether paths can materialize (e.g., BGP communities), and they avoid path computation. Thus, Tiramisu is much faster than state-of-the-art approaches that always consider all attributes (e.g., BGP local preferences) and enumerate actual paths [6, 23] (§8).

TDFS. Finally, *Tiramisu depth-first search* (*TDFS*) is a novel polynomial-time graph traversal algorithm to check for the existence of paths and verify category *iii* policies. *TDFS* makes a constant number of calls to the canonical DFS algorithm. Each call is to a subgraph of the TPG that models the interaction between tag-based (e.g., BGP-community-based) route filters along a path that control if a path can materialize.

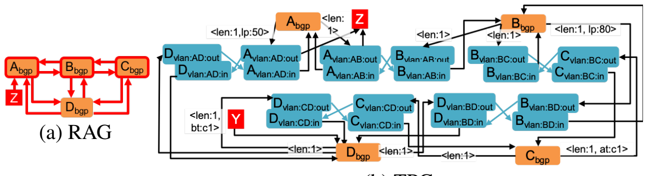
Tiramisu further improves over state-of-the-art verifiers for some category *i* and *ii* policies by avoiding unnecessary path computation. Specifically, we note that some category *i* and *ii* properties require knowing *when* certain paths are taken (i.e., after how many link failures, or after how many more-preferred paths have failed). For such properties, ex-



(a) RAG

(b) TPG

Figure 2: Graphs for network in Figure 1a



(a) RAG

(b) TPG

Figure 3: Graphs for network in Figure 1b

haustively exploring all failures by running *TPVP* for each scenario, while sufficient, is overkill. To avoid enumerating not-so-useful failure scenarios, Tiramisu leverages the graph structure of the network model to run a variant of Yen's dynamic programming based algorithm for k-shortest paths [31], to directly compute a preference order of paths that manifest under arbitrary failures. Our variant, *TYEN*, invokes *TPVP* in a limited fashion from within Yen's execution, minimizing path exploration. For PREF over k paths, we simply use *TYEN* to compute the top-k paths over the TPG. Likewise, we use *TYEN* to accelerate KFAIL, a category *ii* policy. Note that *TYEN* can only be applied to networks whose path metrics are monotonic [16].

4 Tiramisu Graph Abstraction

In this section, we describe in detail the two types of graphs used in Tiramisu: *routing adjacencies graphs* (RAGs) and *traffic propagation graphs* (TPGs). TPGs are partially based on RAGs, and both are based on a network's configurations and physical topology.

4.1 Routing adjacencies graphs

RAGs encode routing adjacencies to allow Tiramisu to determine which routing processes may learn routes to specific IP subnets. Tiramisu constructs a RAG for each of a network's subnets. For example, Figures 2a, and 3a show the RAGs for subnet Z for the networks in Figure 1.

Vertices. A RAG has a vertex for each routing process. For example, B_{bgp} and B_{ospf} in Figure 2a represent the BGP and OSPF processes on router B in Figure 1a. A RAG also has a vertex for each device with static routes for the RAG's subnet.

Edges. A RAG contains an edge for each routing adjacency. Two BGP processes are adjacent if they are explicitly configured as neighbors. Two OSPF process are adjacent if they are configured to operate on router interfaces in the same Layer 2 (L2) broadcast domain (which can be determined from the topology and VLAN configurations). These adjacencies are represented using pairs of directed edges—e.g., $E_{bgp} \leftrightarrows B_{bgp}$ and $B_{ospf} \leftrightarrows C_{ospf}$—since routes can flow between these processes in either direction. However, if two processes are iBGP neighbors then a special pair of directional edges are used—e.g., $B_{bgp} \overset{\leftarrow}{\underset{\dashrightarrow}{}} C_{bgp}$—because iBGP processes do not forward routes learned from other iBGP processes. A routing adjacency is also formed when one process (the redistributor) distributes routes from another process on the same device (the redistributee). This is encoded with a unidirectional edge from redistributee to redistributor. Vertices representing static routes may be redistributees, but will not have any other edges.

Taints. To determine which routing processes may learn routes to specific destinations, Tiramisu runs a "tainting" algorithm on the RAG. All nodes that originate a route for the subnet associated with the RAG (including vertices corresponding to static routes) are tainted. Then taints propagate freely across edges to other vertices, with one exception: when taints traverse an iBGP edge they cannot immediately traverse another iBGP edge. For example, in Figure 2a, E_{bgp} is tainted, because it originates a route for Z. Then taints propagate from E_{bgp} to B_{bgp} to C_{bgp}, but not to D_{bgp}. No OSPF vertices are tainted, because no OSPF processes originate a route for Z and no processes are configured to redistribute routes.

The tainting algorithm assumes all configured adjacencies are active and no routes are filtered. However, for an adjacency to be active in the real network, certain conditions must be satisfied: e.g., B_{ospf} must compute a route to C's loopback interface and vice versa in order for B_{bgp} to exchange routes with C_{bgp}. These dependencies are encoded in TPGs.

4.2 Traffic propagation graph

A process P on router R can advertise a route for subnet S to an adjacent routing process P' on router R' if all of the following dependencies are satisfied: (*i*) P learns a route for S from an adjacent process, or P is configured to originate a route for S; (*ii*) neither P or P' filters the advertisement; (*iii*) another process/switch on R learns a route to R', or R is connected to the same subnet/layer-2 domain as R'; and (*iv*) R is physically connected to R' through a sequence of one or more links. TPGs encode these dependencies. Tiramisu constructs a TPG for every pair of a network's IP subnets. Figures 2b and 3b show the TPGs that model how traffic from subnet Y is propagated to subnet Z for the networks in Figure 1.

Vertices. A TPG's vertices represent the routing information bases (RIBs) present on each router and the (logical) traffic ingress/egress points on each router/switch. Each routing process maintains its own RIB, so the TPG contains a vertex for each process: e.g., B_{bgp} and B_{ospf} in Figure 2b correspond to the BGP and OSPF processes on router B in Figure 1a.

Traffic propagates between routers and switches over VLANs. Consequently, the TPG contains a pair of ingress/egress vertices for each of a device's VLANs: e.g., $S1_{v1:in}$ and $S1_{v1:out}$ in Figure 2b correspond to the VLAN on switch $S1$ in Figure 1a. Tiramisu creates implicit VLANs for pairs of directly connected interfaces: e.g., $B_{vlan:BE:in}$,

$B_{vlan:BE:out}$, $E_{vlan:BE:in}$, and $E_{vlan:BE:out}$ in Figure 2b correspond to the directly connected interfaces on routers B and E in Figure 1a.

The TPG also includes vertices for the source and destination (target) of the traffic being propagated: e.g., Y and Z, respectively, in Figure 2b.

Edges. A TPG's edges reflect the virtual and physical "hops" the traffic may take. Edges model dependencies as follows:

- **Layers 1 & 2:** For each VLAN V on device D, the egress vertex for V on D is connected to the ingress vertex for V on device D' if an interface on D participating in V has a direct physical link to an interface on D' participating in V: e.g., $B_{vlan1:out} \rightarrow S1_{vlan1:in}$ in Figure 2b corresponds to the physical link between B and $S1$ in Figure 1a. Also, the ingress vertex for V on D is connected to the egress vertex for V on D to model L2 flooding: e.g., $S1_{vlan1:in} \rightarrow S1_{vlan1:out}$.
- **OSPF's dependence on L2:** The vertex for OSPF process P on router R is connected to the egress vertex for VLAN V on R if P is configured to operate on V: e.g., $D_{ospf} \rightarrow D_{vlan:CD:out}$ and $D_{ospf} \rightarrow D_{vlan:1:out}$ in Figure 2b model OSPF operating on router D's VLANs in Figure 1a.
- **BGP's dependence on connected and OSPF routes:** For each peer N of the BGP process P on router R, an edge is created from the vertex for P to the egress vertex for VLAN V on R if N's IP address falls within the subnet assigned to V: e.g., $E_{bgp:B} \rightarrow E_{vlan:BE:out}$ in Figure 2b models the BGP process on router E communicating with the adjacent process on directly connected router B in Figure 1a. If no such V exists, then the vertex for P is connected to the vertex for OSPF process P' on R: e.g., $B_{bgp} \rightarrow B_{ospf}$ models the BGP process on B communicating with the adjacent process on router C (which operates on a loopback interface) via an OSPF-computed path.
- **Routes to the destination:** Every VLAN ingress vertex on router R is connected to the vertex for process P on R if the vertex for P in the RAG is tainted: e.g., $B_{vlan:BC:in} \rightarrow B_{bgp}$ in Figure 2b is created due to the taint on B_{bgp} in Figure 2a, which models that fact that the BGP process on B may learn a route to subnet Z from the adjacent process on E. If the destination subnet T is connected to R and at least one routing process on R originates T, then every VLAN ingress vertex on R is connected to the vertex for T: e.g., $E_{vlan:BE:in} \rightarrow Z$ in Figure 2b. If the source subnet S is connected to R and the vertex for process P on R is tainted in the RAG, then the vertex for S is connected to the vertex for P: e.g., $Y \rightarrow C_{bgp}$ in Figure 2b.

Filters. As mentioned in §4.1, a RAG may overestimate which processes learn a route to a subnet due to route and packet filters not being encoded in the RAG. A TPG models filters using two approaches: edge pruning and edge attributes.

Tiramisu uses edge pruning to model prefix- or neighbor-based filters. A BGP process P may filter routes imported from a neighbor P' (or P' may filter routes exported to P) based on the advertised prefix or neighbor from (to) whom the route is forwarded. Tiramisu models such a filter by removing from the vertex associated with P, the outgoing edge that corresponds to P'. For example, if router B in Figure 1a had an import filter, or router E had an export filter, that block routes for Z, then edge $B_{bgp} \rightarrow B_{vlan:BE:out}$ would be removed from Figure 2a. Note that import and export filters are both modeled by removing an outgoing edge from the vertex associated with the importing process. OSPF is limited to filtering incoming routes based on the advertised prefix. Tiramisu models such route filters by removing all outgoing edges from the vertex associated with the OSPF process where the filter is deployed. Lastly, packets sent (received) on VLAN V can be filtered based on source/destination prefix. Tiramisu models such packet filters by removing the outgoing (incoming) edge that represents the physical link connecting V to its neighbor.

Tiramisu uses edge attributes to model tag- (e.g., BGP community- or ASN-) based filters. If a BGP process P filters routes imported from a neighor P' (or P' filters routes exported to P) based on tags, then Tiramisu adds a "blocked tags" attribute to the outgoing edge from the vertex associated with P that corresponds to P'. For example, the edge $D_{bgp} \rightarrow D_{vlan:CD:out}$ in Figure 3b is annotated with $bt = \{c1\}$ to encode the import filter router D applies to routes from C in Figure 1b. Edges can also include "added tags" and "removed tags" attributes: e.g., $C_{bgp} \rightarrow C_{vlan:BC:out}$ is annotated with $at = \{c1\}$ to encode the export filter router B applies to routes advertised to C. Notice that tag actions defined in import and export filters are both added to an outgoing edge from the vertex associated with the importing process.

Costs/preferences. Each routing protocol uses a different set of *metrics* to express link and path costs/preferences. For example, OSPF uses link costs, and BGP uses AS-path length, local preference (lp), etc. Similarly, administrative distance (*AD*) allows routers to choose routes from different protocols. Hence a single edge weight cannot model the route selection decisions of all protocols. Tiramisu annotates the outgoing edges of OSPF, BGP, and VLAN ingress vertices with a *vector of metrics*. Depending on the edge, certain metrics will be null: e.g., OSPF cost is null for edges from BGP vertices.

5 Category (i) Policies

We now describe how Tiramisu verifies policies that require knowing the actual path taken in the network under a given failure (§3.2). One such policy is path preference (PREF). For example, $P_{pref} = p1 \gg p2 \gg p3$, states when path $p1$ fails, $p2$ (if available) should be taken; and when $p1$ and $p2$ both fail, $p3$ (if available) should be taken. A path can become unavailable if a link or device along the path fails. We model such failures by removing edges (between egress and ingress VLAN vertices) or vertices from the TPG. To verify PREF, we need to know what alternate paths materialize, and whether a materialized path is indeed the path meant to be taken according to preference order. Similar requirements arise for verifying multipath consistency (MC).

In this section, we introduce an algorithm, *TPVP*, to compute the actual path taken in the network. *TPVP* can be used to exhaustively explore failures to verify category (i) policies, but it is slow. We show how to accelerate verification using a dynamic-programming based graph algorithm.

5.1 Tiramisu Path Vector Protocol

Griffin et al. observed that BGP attempts to solve the *stable paths problem* and proposed the Simple Path Vector Protocol (SPVP) for solving this problem in a distributed manner [15]. Subsequently, Sobrinho proposed routing algebras for modeling the route computation and selection algorithms of *any* routing protocol in the context of a path vector protocol [25]. Griffin and Sobrinho then extended these algebras to model routing across administrative boundaries—e.g., routing within (using OSPF) and across (using BGP) ASes [16]. However, they did not consider the dependencies between protocols within the same administrative region—e.g., iBGP's dependence on an OSPF. Recently, Plankton addressed these dependencies by solving multiple stable paths problems, and proposed the Restricted Path Vector Protocol (RPVP) for solving these problems in a centralized manner [23].

Below, we explain how the *Tiramisu Path Vector Protocol* (*TPVP*) leverages routing algebras and extends SPVP to compute paths using our graph abstraction. Since SPVP was designed for Layer 3 protocols, *TPVP* works atop a simplified TPG called the Layer 3 TPG (L3TPG). Tiramisu uses path contraction to replace L2 paths with an L3 edge connecting L3 nodes, to model the fact that advertisements can flow between routing processes as long as there is at least one L2 path that connects them. Recall that the incoming edges of VLAN egress vertices and the outgoing edges of VLAN ingress vertices include (non-overlapping) edge labels (§4.2); these are combined and applied to the L3 edge(s) that replace the L2 path(s) containing these L2 edges.

Routing algebras. Routing algebras [16, 25] model routing protocols' path cost computations and path selection algorithms. An algebra is a tuple $(\Sigma, \preceq, L, \oplus, O)$ where:

- Σ is a set of *signatures* representing the multiple metrics (e.g., AS-path length, local pref, ...) associated with a path.
- $\preceq$ is the *preference* relation over signatures. It models route selection, ranking paths by comparing multiple metrics of multiple path signatures in a predefined order (e.g., first compare local pref, then AS-path length, ...)
- L is set of *labels* representing multi-attribute edge-weight.
- $\oplus$ is a function $L \times \Sigma \rightarrow \Sigma$, capturing how labels and signatures combine to form a new signature; i.e., $\oplus$ models path cost computations. $\oplus$ has multiple operators, each computing on certain metrics.[2]
- O is the signature attached to paths at origination.

In Tiramisu, path signatures contain metrics from all possible protocols (e.g., OSPF cost, AS-path length, AD, ...), but $\preceq$ and $\oplus$ are defined on a per-protocol basis and only operate on the metrics associated with that protocol. For example, $\oplus_{BGP}$ sets local pref and adds AS-path lengths from a label ($\lambda \in L$), but copies the OSPF cost and AD directly from the input signature ($\sigma \in \Sigma$) to the output signature ($\sigma' \in \Sigma$). Similarly, $\preceq_{BGP}$ compares local pref, AS-path lengths, and OSPF link costs[3] but does not compare AD.

TPVP. *TPVP* (Algorithm 1) is derived from SPVP [15]. For each vertex in the L3TPG, *TPVP* computes and selects a most-preferred path to *dst* based on the (signatures of) paths selected by adjacent vertices. However, *TPVP* extends *SPVP* in two fundamental ways: (*i*) it uses a shared memory model instead of message passing, akin to *RPVP* [23]; and (*ii*) it models multiple protocols in tandem by computing path signatures and selecting paths using routing algebra operations corresponding to different protocols: e.g., $\oplus_{BGP}$ and $\preceq_{BGP}$ are applied at vertices corresponding to BGP processes.

For each peer v of each vertex u (v is a peer of u if $u \rightarrow v \in$ L3TPG) *TPVP* uses variables $p_u(v)$ and $\sigma_u(v)$ to track the most preferred path to reach *dst* through v and the path's signature, respectively. Likewise, variables p_u and σ_u represent the most preferred path and its signature to reach *dst* from u.

In the initial state, *TPVP* sets the *path* and *sign* values of all nodes except *dst* to null (line 2). p_{dst} is set to ε and σ_{dst} is set to O, since it "originates" the advertisement (line 3). Similar to *SPVP*, there are three steps in each iteration. First, for each node u, *TPVP* computes all its $p_u(*)$ and $\sigma_u(*)$ values based on the path signatures of its neighbors and outgoing edge labels $\lambda_{u \rightarrow *}$ (lines 7–10). It calculates the best path based on the preference relation (line 11). If the current p_u changes from previous iteration, then the network has not converged and the process repeats (lines 12–13).

Theorem 1. *If the network control plane converges,* TPVP *always finds the exact path taken in the network under any given failure scenario.*

We prove Theorem 1 is in Appendix C.1. The proof shows that *TPVP* and the TPG correctly model the permitted paths and ranking function to solve the stable paths problem.

Plankton [23] leverages basic SPVP to model the network. But because basic SPVP cannot directly model iBGP, to verify networks that use iBGP, Plankton runs multiple SPVP instances. As mentioned in §2.1, BGP routing depends on the outcome of OSPF routing. Hence, Plankton runs SPVP multiple times: first for multiple OSPF instances, and then for dependent BGP instances. In contrast, because Tiramisu's *TPVP* is built using routing algebra atop a network model with rich edge attributes, we can bring different dependent routing protocols into one common fold of route computation. Thus, we can analyze iBGP networks, and, generally, dependent protocols, "in one shot" by running a single *TPVP* instance.

[2]For example, *ADD* operator adds OSPF link costs and AS-path lengths. *LP* operator sets local pref, *TAGS* operator prohibits paths with a tag, etc.

[3]OSPF cost is used as a tie-breaker in BGP path selection [10].

Algorithm 1 Tiramisu Path Vector Protocol

```
1:  procedure $TPVP(L3TPG)$
2:      $\forall i \in \{V - dst\} : p_i = \varnothing, \sigma_i = \phi$
3:      $p_{dst} = [dst], \sigma_{dst} = O$                ▷ $dst$ originates the route
4:      $converged = false$
5:      while $\neg converged$ do
6:          $converged = true$
7:          for each $u \in L3TPG$ do
8:              for each $v \in peers(u)$ do
9:                  $p_u(v) = edge_{u \to v} \circ p_v$            ▷ add edge to path
10:                 $\sigma_u(v) = \lambda_{u \to v} \oplus_{type(u)} \sigma_v$
11:                 compute $p_u$ and $\sigma_u$ using $\preceq$ over $\sigma_u(*)$
12:                 if $p_u$ has changed then
13:                     $converged = false$
```

In Appendix D, we show that *TPVP* can verify other policies, like "Multipath Consistency (MC)", that requires materialization of certain paths.

5.2 Tiramisu Yen's algorithm

To verify PREF, Tiramisu runs *TPVP* multiple times to compute paths for different failure scenarios. For example, while verifying P_{pref}, Tiramisu runs *TPVP* for all possible failures (edge removals) that render paths $p1$ and $p2$ unavailable. Then, it checks if *TPVP* always computes $p3$ (if available). While correct, this is tedious and slow overall.

We can accelerate the verification of this policy by leveraging the graph structure of the L3TPG and developing a graph algorithm that avoids unnecessary path explorations/computations. Specifically, we observed that there are similarities between analyzing PREF and finding the k shortest paths in a graph [31]. This is because, in the k shortest paths problem, the k^{th} shortest path is taken only when $k-1$ paths have failed. To avoid enumerating all possible failures of all $k-1$ shorter paths, Yen [31] introduced an efficient algorithm for this problem that uses dynamic programming to avoid failure enumeration. Yen uses the intuition that the k^{th} shortest path will be a small perturbation of the previous $k-1$ shortest paths. Instead of searching over the set of all paths, which is exponential, Yen constructs a polynomial candidate set from the previous $k-1$ paths, in which the k^{th} path will be present.

To accelerate PREF, our *TYEN* algorithm makes two simple modifications to Yen. Yen uses Dijkstra to compute the shortest path. We replace Dijkstra with *TPVP*. Next, we add a condition to check that during the i^{th} iteration, the i^{th} computed path follows the preference order specified in PREF. The detailed description of Yen and *TYEN* are in Appendix A. Note that Yenand hence TYEN, assumes the path-cost composition function is strictly monotonic. Hence, TYEN acceleration can be leveraged only on monotonic networks.

6 Category (ii) Policies

We now describe how Tiramisu verifies policies pertaining to quantitative path metrics, e.g., KFAIL and BOUND. For such policies, Tiramisu uses property-specific ILPs. These ILPs run fast because they abstract the underlying TPG and only model protocol attributes that impact whether paths can materialize (e.g., communities).

6.1 Tiramisu Min-cut

KFAIL states that *src* can reach *dst* as long as there are $< K$ link failures. ARC [14] verifies this policy by computing the min-cut of a graph; if min-cut is $\geq K$, then KFAIL is satisfied.

However, standard min-cut algorithms do not consider how route tags (e.g., BGP communities) impact the existence of paths. For example, in Figure 3b, any path that includes $D_{bgp:c} \to D_{vlan:CD:out}$ followed by $C_{bgp:b} \to C_{vlan:BC:out}$ is prohibited due to the addition and filtering of tags on routers B and D, respectively. In this manner, tags prohibit paths with certain combinations of edges in the TPG, making the TPG a "correlated network". It is well known that finding the min-cut of a correlated network is NP-Hard [30]. Note that traffic flows in the direction opposite to route advertisements. Hence, the prohibited paths have tag-blocking edges followed by tag-adding edges.

We propose an ILP which accounts for route tags, but ignores irrelevant edge attributes (e.g., edge costs), to compute the min-cut of a TPG For brevity, we explain the constraints at a high-level, leaving precise definitions to Appendix B.1. Equation numbers below refer to equations in Appendix B.1.

The objective of the ILP is to minimize the number of physical link failures (F_i) to disconnect *src* from *dst*.

$$\textbf{Objective:} \qquad \texttt{minimize} \sum_{i \in pEdges} F_i \qquad (1)$$

Traffic constraints. We first define constraints on reachability. The base constraint states that *src* originates the traffic (Eqn 2). To disconnect the graph, the next constraint states that the traffic must not reach *dst* (Eqn 3). For other nodes, the constraint is that traffic can reach a node if it gets *propagated* on any of its incoming edges (Eqn 4).

Now we define constraints on traffic propagation. Traffic can propagate through an edge e if: it reaches the start node of that edge; the traffic does not carry a tag that is blocked on that edge; and, if the edge represents an inter-device edge, the underlying physical link has not failed. This is represented as shown in Eqn 5.

Tags. We now add constraints to model route tags. The base constraints state that each edge that blocks on a tag forwards that tag, and each edge that removes that tag does not forward it further (Eqn 6 and Eqn 7). For other edges, we add the constraint that edge e forwards a tag t iff the start node of edge e receives traffic with that tag (Eqn 8). Finally, we add the constraint that an edge e carries a blocked tag iff that blocked tag can be added by edge e (Eqn 9).

We prove the correctness of this ILP in Appendix C.2 based on the correctness of Tiramisu's modeling of permitted paths.

Using TYEN for KFAIL with ACLs. This ILP is not accurate when packet filters (ACLs) are in use. ACLs do not influence advertisements. Hence, routers can advertise routes for traffic that get blocked by ACLs. Recall that during graph creation, Tiramisu removes edges that are blocked by ACLs §4.2. This leads to an incorrect min-cut computation as shown below:

Assume *src* and *dst* are connected by three paths $P1$, $P2$ and $P3$ in decreasing order of path-preference. Also assume these paths are edge disjoint and path $P2$ has a data plane ACL on it. If a link failure removes $P1$, then the control plane would select path $P2$. However, all packets from *src* will be dropped at the ACL on $P2$. In this case, a single link failure (that took down $P1$) is sufficient to disconnect *src* and *dst*. Hence the true min-cut is 1. On the other hand, Tiramisu would remove the offending ACL edge from the graph. The graph will now have paths $P1$ and $P3$, and the ILP would conclude that the min-cut is 2, which is incorrect.

We address this issue as follows. Nodes can become unreachable when failures either disconnect the graph or lead to a path with an ACL. We compute two quantities: (1) N: Minimum failures that disconnects the graph, and, (2) L: Minimum failures that cause the control plane to pick a path blocked by an ACL. The true min-cut value is $min(N, L)$.

First we use our min-cut ILP to compute N. To compute L, in theory, we could simply run TPVP to exhaustively explore k-link failures for $k = 1, 2, ..$, and determine the smallest failure set that causes the path between *src* and *dst* to traverse an ACL. However, this is expensive.

We can accelerate this process by leveraging *TYEN*, similar to our approach for PREF. We first construct a TPG without removing edges for ACLs. Then, we run *TYEN* until we find the first path with a dropping ACL on it. Say this was the M^{th} preferred path. Then, we remove all edges from the graph that do not exist in any of the previous $M-1$ paths. Next, we use our min-cut ILP to compute the minimal failures L to disconnect this graph. This represents the minimal failures to disconnect previous $M-1$ paths and pick the ACL-blocked path. If $min(L, N) \geq K$ then KFAIL is satisfied.

Overall, to compute KFAIL, the above requires Tiramisu to run (a) *TYEN* to compute M paths; and (b) two min-cut ILPs to compute N and L respectively. Note that to verify KFAIL, Minesweeper will explore all combinations of k link failures.

6.2 Tiramisu Longest path

Always bounded length policy (BOUND) states that for a given K, BOUND is true if under every failure scenario, traffic from *src* to *dst* never traverses a path longer than K hops. Enumerating all possible paths and finding their length is infeasible.

However, this policy can be verified efficiently by viewing it as a variation of computing a quantitative path property, namely the *longest path problem*: for a given K, BOUND is true if the longest path between *src* and *dst* is $\leq K$.

Finding the longest path between two nodes in a graph is also NP hard [19]. To verify BOUND, we thus propose another ILP whose objective is to maximize the number of inter device edges (*dEdges*) traversed by traffic (A_i).

$$\textbf{Objective:} \quad \texttt{maximize} \sum_{i \in dEdges} A_i \qquad (2)$$

We present detailed constraints and a proof of correctness in Appendices B.2 and C.2, respectively.

Constraints. We first add constraints to ensure that traffic flows on one path, *src* sends traffic, and *dst* receives it (Eqn 11 and Eqn 12). For other nodes, we add the flow conservation property, i.e., the sum of incoming flows is equal to the sum of outgoing flows (Eqn 13). Finally, we add constraints on traffic propagation: traffic will be blocked on edge e if it satisfies the tag constraints (Eqn 14).

In Appendix D, we show that a similar ILP can be used to verify other policies of interest—e.g., all paths between *src* and *dst* have equal length (EB).

7 Category (iii) Policies

Finally, we describe how Tiramisu verifies policies that only require us to check for just the existence of a path, e.g., "always blocked" (BLOCK). For these policies, we use a new simple graph traversal algorithm. Tiramisu's performance is fastest when checking for such policies (§8).

Standard graph traversal algorithms, like *DFS*, also do not account for tags. DFS will identify the prohibited path from Figure 3b ($D_{bgp:c} \rightarrow D_{vlan:CD:out}$ followed by $C_{bgp:b} \rightarrow C_{vlan:BC:out}$) as valid, which is incorrect. To support tags, we propose *TDFS*, Tiramisu Depth First Search (Algorithm 2). *TDFS* makes multiple calls to *DFS* to account for tags.

TDFS. As mentioned in §4, edges can add, remove or block on tags. In presence of such edges, the order in which these edges are traversed in a path determines if *dst* is reachable.

TDFS first checks if *dst* is unreachable from *src* according to *DFS* (line 3, 4). If they are reachable, then *TDFS* checks if all paths that connect *src* to *dst* (i) have an edge (say X) that blocks route advertisements and hence traffic (for *dst*) with a tag (line 5 to 6), (ii) followed by an edge (say Y) that adds the tag to the advertisements for *dst* (line 7 to 8), and (iii) has no edge between X and Y that removes the tag (line 9 to 10). If all these conditions are satisfied, then *src* and *dst* are unreachable. If any of these conditions are violated, the nodes are reachable.

We prove the correctness of *TDFS* in Appendix C.3.

The above algorithm naturally applies to verifying BLOCK. It can similarly be used to verify WAYPT ("always waypointing"): after removing the waypoint, if *src* can reach *dst*, then there is a path that can reach *dst* without traversing the waypoint. *TDFS* can also verify "always chain of waypoints (WAYPT)" and "no blackholes (BH)" (Appendix D).

8 Evaluation

We implemented Tiramisu in Java (≈ 7K lines of code) [2]. We use Batfish [13] to parse router configurations and Gurobi [3] to solve our ILPs. We evaluate Tiramisu on a variety of issues:

- How quickly can Tiramisu verify different policies?
- How does Tiramisu perform compared to state-of-the-art?

Algorithm 2 Always blocked with tags

```
1:  procedure $TDFS(G, src, dst)$
2:      $tA, tR, tB \leftarrow$ edges that add, remove, or block on tag (respectively)
3:      if $dst$ is unreachable by $DFS$ (G, $src$) then
4:          return true (nodes are already unreachable)
5:      if $dst$ is reachable by $DFS$ (G-$tB$, $src$) then
6:          return false ($\exists$ path $src \rightsquigarrow dst$ where tagged-routes are not blocked)
7:      if $\exists e_b \in tB$ s.t. $dst$ is reachable by $DFS$ (G-$tA$, $e_b$) then
8:          return false (tag-blocking edges can get ads for $dst$ without tags)
9:      if $\forall e_b \in tB, e_a \in tA$: $e_a$ is unreachable by $DFS$ (G-$tR$, $e_b$) then
10:         return false (tags always removed before reaching blocking edges)
11:     return true
```

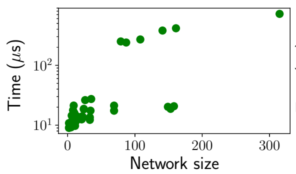


Figure 4: Graph generation time (all networks)

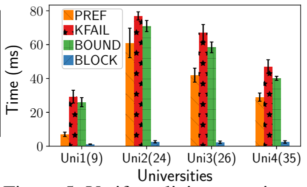


Figure 5: Verify policies on university configs

• How does Tiramisu's performance scale with network size? Our experiments were performed on machines with dual 10-core 2.2 GHz Intel Xeon Processors and 192 GB RAM.

8.1 Network Characteristics

In our evaluation, we use configurations from (a) 4 real university networks, (b) 34 real datacenter networks operated by a large online service provider, (c) 7 networks from the topology zoo dataset [20], and (d) 5 networks from the Rocketfuel dataset [27]. The university networks have 9 to 35 devices and are the richest in terms of configuration constructs. They include eBGP, iBGP, OSPF, static routes, packet/route filters, BGP communities, local preferences, VRFs and VLANs. The datacenter networks have 2 to 24 devices and do not employ local preference or VLANs. The topology zoo networks have 33 to 158 devices, and the Rocketfuel networks have 79 to 315 devices. The configs for topology zoo and Rocketfuel were synthetically generated for random reachability policies [11, 23] and do not contain static routes, packet filters, VRFs or VLANs. Details on the TPGs for these networks are in Appendix E. The main insight is that the number of routing processes and adjacencies per device varies. Hence, the number of nodes and edges in the TPG do not monotonically increase with network size.

Policies. We consider five polices: (PREF) path preference, (KFAIL) always reachable with $< K$ failures, (BOUND) always bounded length, (WAYPT) always waypointing, and (BLOCK) always unreachable. Recall that: PREF is *category i* and is accelerated by *TYEN* calling *TPVP* from within; KFAIL and BOUND are *category ii* and use ILPs; BLOCK and WAYPT are *category iii* and use *TDFS*.

8.2 Verification Efficiency

We examine how efficiently Tiramisu can construct and verify these TPGs. First, we evaluate the time required to generate the TPGs. We use configurations from all the networks. Figure 4 shows the time taken to generate a *traffic class-specific TPG* for all networks. Tiramisu can generate these graphs, even for large networks, in ≈ 1 *ms*.

Next, we examine how efficiently Tiramisu verifies various policies. Since the university networks are the richest in terms of configuration constructs, we use them in this experiment. Because of VRF/VLAN, Minesweeper, Plankton and ARC cannot model these networks. Figure 5 shows the time taken to verify PREF, KFAIL, BOUND, and BLOCK. Since WAYPT uses *TDFS*, it is verified in a similar order-of-magnitude time as BLOCK. Hence for brevity, we do not show their results. In this and all the remaining experiments, the values shown are the median taken over *100 runs* for *100 different traffic classes*. Error bars represent the std. deviation.

We observe that BLOCK can be verified in less than **3 ms**. Since it uses a simple graph traversal algorithm (*TDFS*), it is the fastest to verify among all policies. In fact, for BLOCK, our numbers are comparable to ARC [14]. The time taken to verify PREF is higher than BLOCK, because *TPVP* and *TYEN* algorithms are more complex, as they run our path vector protocol to convergence to find paths (and in *TYEN*'s case, *TPVP* is invoked several times). Finally, KFAIL and BOUND, both use an ILP and are the slowest to verify. However, they can still be verified in $\approx$ **80 ms** per traffic class.

Although $Uni2$ and $Uni3$ have fewer devices than $Uni4$, their TPGs are larger (§E), so it takes longer to verify policies.

8.3 Comparison with Other tools

Next, to put our performance results in perspective, we compare Tiramisu with other state-of-art verification tools.

Minesweeper [6] In this experiment we use datacenter networks and consider policies PREF, BOUND, WAYPT, and BLOCK. Minesweeper takes the number of failures (K) as input and checks if the policy holds as long as there are $\leq K$ failures. To verify a property under all failure scenarios, we set the value of K to one less than the number of physical links in the network. Figure 6 shows the time taken by Tiramisu and Minesweeper to verify these policies, and Figure 7 shows the speedup provided by Tiramisu.

PREF is the only policy where speedup does not increases with network size. This is because larger networks have longer path lengths and more possible candidate paths, both of which affect the complexity of the *TYEN* algorithm. The number of times *TYEN* invokes *TPVP* increases significantly with network size. Hence the speedup for PREF is relatively less, especially at larger network sizes. For BOUND, the speedup is as high as $50X$. For policies that use *TDFS* (BLOCK and WAYPT), Tiramisu's speedup is as high as 600-800X.

Next, we compare the performance of Tiramisu and Minesweeper for the same policies but without failures, e.g. "currently reachable" instead of "always reachable". Tiramisu verifies these policies by generating the actual path using *TPVP*. Figure 8 (a, b, c, and d) shows the speedup provided by Tiramisu for each of these policies. Even for *no failures*,

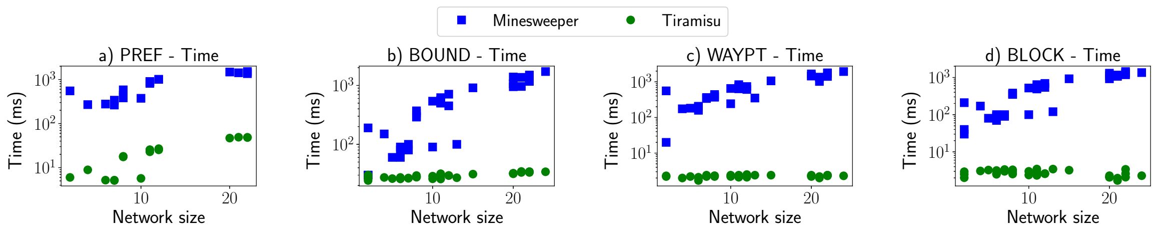


Figure 6: Performance under all failures: Tiramisu vs Minesweeper (datacenter networks)

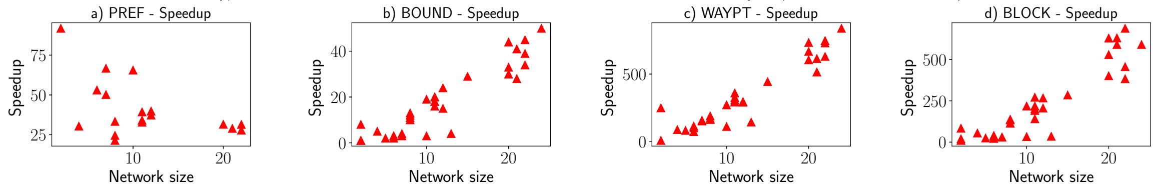


Figure 7: Speedup under all failures: Tiramisu vs Minesweeper (datacenter networks)

Tiramisu significantly outperforms Minesweeper across all policies. Minesweeper has to invoke the SMT solver to find a satisfying solution even in this simple case.

To shed further light on Tiramisu's benefits w.r.t. Minesweeper, we compare the number of variables used by Minesweeper's SMT encoding and Tiramisu's ILP encoding to verify KFAIL and BOUND. We observed that Tiramisu uses 10-100X fewer variables than Minesweeper. For Tiramisu, we also found that BOUND uses fewer variables than KFAIL.

Plankton [23] Next, we compare Tiramisu against Plankton using the Rocketfuel topologies. We generate two sets of configurations: one with only OSPF and one with iBGP and OSPF. We run Plankton with 32 cores to verify reachability with one failure, i.e. k=1 in KFAIL. Figure 9(a) shows the time taken by Plankton (SPVP on 32 cores) and Tiramisu (ILP on 1 core) to verify this policy. Due to dependencies, Plankton (P-iBGP) performs poorly for iBGP networks. It gave an out-of-memory error for large networks. For small networks, Tiramisu (T-iBGP) outperforms Plankton by 100-300X. On networks that run only OSPF, Plankton (P-OSPF) performed better on a single network with 108 routers. Communication with the authors revealed that Plankton may have gotten lucky by quickly trying a link failure that violated the policy. Disregarding that anomaly, Tiramisu (T-OSPF) outperforms Plankton by 2-50X on the OSPF-only networks.

Batfish [13] Data-plane verifiers, like Batfish, generate data planes and verify policies on each generated data plane. ARC [14] showed that Batfish is impractical to use even for small failures. Here, we show that even without failures, Tiramisu outperforms Batfish. We run Batfish with all its optimization, on the datacenter networks to verify reachability without failures. As mentioned earlier, Tiramisu verifies this policy using *TPVP*. Figure 9(b) shows that it outperforms Batfish by 70-100X.

Bonsai Bonsai [7] introduced a compression algorithm to improve scalability of configuration verifiers like Minesweeper, to verify certain policies exclusively under **no failures**. We repeat the previous experiment to evaluate Bonsai (built on Minesweeper). Figure 9(b) shows that Tiramisu still outperforms Bonsai, and can provide speedup as high as 9X.

8.4 Scalability

Here, we evaluate Tiramisu's performance to verify PREF, KFAIL, BOUND, and, BLOCK, on large networks from the topology zoo. Figure 10 shows the time taken to verify these policies. Tiramisu can verify these policies in $<$ **0.12 s**.

For large networks, time to verify PREF (*TYEN*) is as high as BOUND. Again, this is due to larger networks having longer and more candidate paths. Large networks also have high diversity in terms of path lengths. Hence, we see more variance in the time to verify PREF compared to other policies.

For large networks, the time to verify KFAIL is significantly higher than other policies. This happens because KFAIL's ILP formulation becomes more complex, in terms of number of variables, for such large networks. As expected, verifying BLOCK is significantly faster than all other policies, and it is very fast across all network sizes.

Impact of *TYEN* acceleration. In our analysis of PREF and KFAIL, we invoke *TYEN*'s acceleration. To evaluate how much acceleration *TYEN* provided, we now measure the time taken to verify PREF on the Rocketfuel and datacenter networks, with and without *TYEN*'s optimization. Our main conclusions are (i) on small networks (< 20 devices), *TYEN* provides a acceleration as high as 1.4X, and (ii) on large networks (> 100 devices), *TYEN* provides acceleration as high as 3.8X.

Evaluation Summary. By decoupling the encoding from algorithms, Tiramisu uses custom property-specific algorithms with graph-based acceleration to achieve high performance.

9 Extensions and limitations

Although we describe Tiramisu's graphs in the context of BGP and OSPF, the same structure can be used to model other popular protocols (e.g., RIP and EIGRP). Additionally, virtual routing and forwarding (VRFs) can be modeled by replicating routing process vertices for each VRF in which the process participates. To verify policies for different traffic classes, Tiramisu generates a TPG per traffic class. To reduce the number of TPGs, we can compute non-overlapping packet equivalence classes (PEC) [17, 23] and create a TPG per PEC.

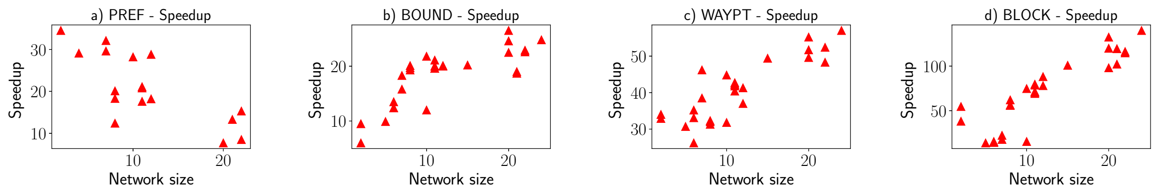


Figure 8: Speedup under no failures: Tiramisu vs Minesweeper (datacenter networks)

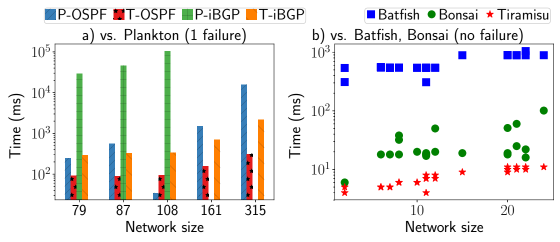


Figure 9: Comparisons with Plankton, Batfish, and Bonsai

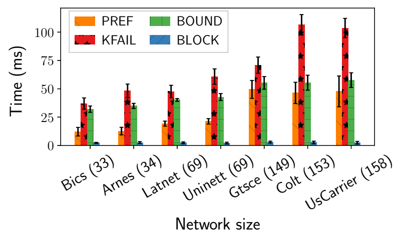


Figure 10: Performance on scale (topology zoo)

Unlike SMT-based tools, Tiramisu does not symbolically model advertisements. Consequently, Tiramisu cannot determine if there exists some external advertisement that could lead to a policy violation; Tiramisu can only exhaustively explore link failures. Tiramisu must be provided concrete instantiations of external advertisements; in such a case, Tiramisu can analyze the network under the given advertisement(s) and determine if any policies can be violated. A related issue is that Tiramisu cannot verify control plane equivalence: two control planes are equivalent, if the behavior of the control planes (paths computed) is the same under all advertisements and all failure scenarios. In essence, while Tiramisu can replace Minesweeper for a vast number of policies, it is not a universal replacement. Minesweeper's SMT-encoding is useful to explore advertisements. Additionally, Tiramisu cannot check quantitative advertisement policies—e.g., does an ISP limit the number of prefixes accepted from a peer.

Tiramisu's modeling of packet filters in the TPG only considers IP-based filtering. Consequently, in networks with protocol- or port-based packet filters, Tiramisu may over- or under-estimate reachability for packets using particular ports or protocols. Additionally, Tiramisu does not account for packet filters that impact route advertisements—e.g., filtering packets destined for a particular BGP neighbor—or route filters where multiple tags affect the same destination—e.g., community groups, AS-path filters, etc. We plan to extend Tiramisu in the future to address these limitations.

Finally, Tiramisu cannot correctly model a control plane where (1) iBGP routes lead to route deflection, and (2) an iBGP process assigns preferences (lp) or tag-based filters to routes received from its iBGP neighbors (Appendix C).

10 Related Work

We surveyed various related works in detail in earlier sections. Here, we survey others that were not covered earlier.

ERA [12] is another control plane verification tool. It symbolically represents advertisements which it propagates through a network and transforms it based on how routers are configured. ERA can verify reachability against arbitrary external advertisements, but it does not have the full coverage of control plane constructs as Tiramisu to analyze a range of policies. Bagpipe [29] is similar in spirit to Minesweeper and Tiramisu, but it only applies to a network that only runs BGP. FSR [28] focuses on encoding BGP path preferences.

Batfish [13] and C-BGP [24] are control plane simulators. They analyze the control plane's path computation as a function of a given environment, e.g., a given failure or an incoming advertisement, by conducting low level message exchanges, emulating convergence, and creating a concrete data plane. Tiramisu also conducts simulations of the control plane; but, for certain policies, Tiramisu can explore multiple paths at once via graph traversal and avoid protocol simulation. For other policies, Tiramisu only simulates a path vector protocol. Although P-Rex [18] is modeled for fast verification under failures, it focuses solely on MPLS.

11 Conclusion

While existing graph-based control plane abstractions are fast, they are not as general. Symbolic and explicit-state model checkers are general, but not fast. In this paper, we showed that graphs can be used as the basis for general and fast network verification. Our insight is that, rich, multi-layered graphs, coupled with algorithmic choices that are customized per policy can achieve the best of both worlds. Our evaluation of a prototype [2] shows that we offer 2-600X better speed than state-of-the-art, scale gracefully with network size, and model key features found in network configurations in the wild.

Acknowledgments. We thank the anonymous reviewers for their insightful comments. We also thank the network engineers that provided us real university and datacenter configurations. This work is supported by the National Science Foundation grants CNS-1637516 and CNS-1763512.

References

[1] Route selection in cisco routers. https://bit.ly/2He9zYk.

[2] Tiramisu source code. https://github.com/anubhavnidhi/batfish/tree/tiramisu.

[3] Gurobi. http://www.gurobi.com/, 2017.

[4] Widespread impact caused by Level 3 BGP route leak. https://dyn.com/blog/widespread-impact-caused-by-level-3-bgp-route-leak/, 2017.

[5] Verizon BGP misconfiguration creates blackhole. https://www.theregister.co.uk/2019/06/24/verizon_-bgp_misconfiguration_cloudflare/, 2019.

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, 2017.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *ACM SIGCOMM*, 2018.

[8] Theophilus Benson, Aditya Akella, and David Maltz. Unraveling the complexity of network management. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[9] Theophilus Benson, Aditya Akella, and Aman Shaikh. Demystifying configuration challenges and trade-offs in network-based ISP services. In *ACM SIGCOMM*, 2011.

[10] Cisco Systems. BGP best path selection algorithm. http://cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/13753-25.html.

[11] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical network-wide configuration synthesis with autocompletion. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[12] Seyed Kaveh Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd D. Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[13] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*, 2016.

[15] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking (ToN)*, 10(2):232–243, 2002.

[16] Timothy G Griffin and Joäo Luís Sobrinho. Metarouting. In *ACM SIGCOMM computer communication review*, volume 35, pages 1–12. ACM, 2005.

[17] Alex Horn, Ali Kheradmand, and Mukul R Prasad. Delta-net: Real-time network verification using atoms. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[18] Jesper Stenbjerg Jensen, Troels Beck Krøgh, Jonas Sand Madsen, Stefan Schmid, Jiří Srba, and Marc Tom Thorgersen. P-rex: Fast verification of mpls networks with multiple link failures. In *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[19] David Karger, Rajeev Motwani, and GDS Ramkumar. On approximating the longest path in a graph. In *Workshop on Algorithms and Data structures*, pages 421–432. Springer, 1993.

[20] Simon Knight, Hung X Nguyen, Nick Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[21] David A. Maltz, Geoffrey Xie, Jibin Zhan, Hui Zhang, Gísli Hjálmtýsson, and Albert Greenberg. Routing design in operational networks: A look from the inside. In *ACM SIGCOMM*, 2004.

[22] John Moy. RFC2328: OSPF version 2, 1998.

[23] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[24] Bruno Quoitin and Steve Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE network*, 19(6):12–19, 2005.

[25] Joao L Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Transactions on Networking*, 13(5):1160–1173, 2005.

[26] Joao Luis Sobrinho. Network routing with path vector protocols: Theory and applications. In *ACM SIGCOMM*, 2003.

[27] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring isp topologies with rocketfuel. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 133–145. ACM, 2002.

[28] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. FSR: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking (ToN)*, 20(6):1814–1827, 2012.

[29] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016.

[30] Song Yang, Stojan Trajanovski, and Fernando A Kuipers. Optimization problems in correlated networks. *Computational social networks*, 3(1):1, 2016.

[31] Jin Y Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

A Yen's and *TYEN* Algorithm

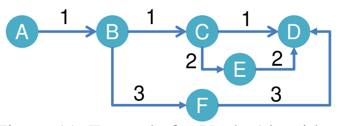


Figure 11: Example for Yen's Algorithm

Yen's algorithm. We use the graph in Figure 11 to explain this algorithm. Although line numbers below refer to *TYEN*, those lines of code also apply to the original Yen's algorithm.

Yen uses two lists: *listA* (to keep track of the shortest path seen so far) and *listB* (to keep track of candidates for the next shortest path). At the start, Yen finds the first shortest path (line 2) from *src* to *dst* using any shortest path algorithm (e.g. Dijkstra's). In Figure 11, it is $A \to B \to C \to D$. Let this path be P. Yen adds P to *listA*.

During each iteration of k, Yen takes every node n in path P (line 13, 14), and finds the *rootPath* and *spurPath* of that node. The *rootPath* of n is the subpath of P from *src* to node n (line 15). The *spurPath* is the shortest path from node n to *dst* (line 20) after making the following changes to the graph: i) to avoid loops, Yen removes all *rootPath* node except n from the graph, ii) to avoid recomputation, Yen removes all outgoing edges e from n, where e is part of any path P_A from *listA* having the same *rootPath* (line 17, 18). For example, if P_A is $A \to B \to C \to D$ and B is n, then $A \to B$ is the *rootPath* and e is $edge_{B \to C}$. By condition (i), Yen removes A and $edge_{A \to B}$. By condition (ii), it removes $edge_{B \to C}$. Then, it computes *spurPath* as $B \to F \to D$. Yen combines the *rootPath* and *spurPath* to form a new path P' (line 21) that is not blocked by tags. P' is added to *listB* if it doesn't already exist in *listA* or *listB* (line 23).

After traversing each node n in path P, Yen picks the shortest path from *listB* and reruns the previous steps with this path as the new P. This continues till Yen finds k paths.

Tiramisu Yen. To verify PREF, we make two simple modifications to Yen (*TYEN*). First, we replace Dijkstra with *TPVP*. Next, we add a condition to check that during each i^{th} iteration of k, the i^{th} path follows the preference order specified in PREF. To achieve this, *TYEN* associates each path with a variable, *eRemoved*. *eRemoved* keeps track (line 22) of edges that were removed from L3TPG (line 17-19) to compute that path. During each iteration of P, *TYEN* identifies the most preferred path specified in PREF that did not have an edge in *P.eRemoved* (line 8). If it varies from P, then preference is violated (line 9, 10).

Algorithm 3 Tiramisu Yen

```
Input:
  $G$ is the graph
  $src$, $dst$ are source and destination nodes
  $K$ is no. of path specified in path-preference policy
  PREF, a map of preference level and path
1: procedure TYEN ($G, src, dst, K$)
2:   $P \leftarrow$ path from src to dst returned by TPVP
3:   $pathsSeen \leftarrow 0$
4:   $P.eRemoved \leftarrow$ [], as best path requires no edge removal
5:   $listA \leftarrow$ [], tracks paths already considered as $P$
6:   $listB \leftarrow$ [], tracks paths not yet considered
7:   while $pathsSeen < K$ do
8:     $mostPref \leftarrow$ most preferred path in PREF whose edges don't overlap with $P.eRemoved$
9:     if $P \neq mostPref$ then
10:      return false, since path preference is violated
11:    $pathsSeen \leftarrow pathsSeen + 1$
12:    add $P$ to $listA$
13:    for i $\leftarrow$ 0 to P.length - 1 do
14:      $sNode \leftarrow i^{th}$ node of P
15:      $rootPath \leftarrow$ subpath of P from $src$ to $sNode$
16:      for each $sp \in listA$ do                ▷ paths in $listA$
17:        if $sp$ has same $rootPath$ at $sNode$ then
18:          remove out edge of $sNode$ in $sp$, so path $sp$ is not considered
19:      remove all nodes and edges of $rootPath$ except $sNode$ to avoid loops
20:      $spurPath \leftarrow$ path from $sNode$ to $dst$ returned by TPVP
21:      $P' \leftarrow rootPath + spurPath$
22:      $eRemoved \leftarrow$ all edges removed in this iteration
23:      add $P'$ to end of $listB$ if $P'$ is valid and $P' \notin [listA, listB]$
24:      append $P'.eRemoved$ to include $eRemoved$
25:      add back all nodes and edges to the graph
26:    P $\leftarrow$ remove first path from $listB$
27:  return true, since loop didn't find preference violation
```

B Tiramisu ILPs

Table 2 lists the boolean indicator variables and functions used in the ILPs.

B.1 Tiramisu Min-cut

We now present the complete ILP for KFAIL (§6.1). We repeat the description of the constraints to ensure ease of reading.

The objective is to minimize the number of physical link

	Name	Description
Variable	F_e	set as 1 if edge e fails
	A_e	set as 1 if traffic propagates on edge e
	R_n	set as 1 if traffic reaches node n
	B_e	set as 1 if edge e carries blocked tag
	$T_{n,t}$	set as 1 if node n forwards tag t
Function	$nodes$	returns all nodes of graph
	$edges$	returns all edges of graph
	$dEdges$	returns all inter-device edges of graph
	$pEdges$	returns all physical edges of the network
	$oNodes$	returns all nodes except src and dst
	$iE(n)$	returns incoming edges of node n
	$oE(n)$	returns outgoing edges of node n
	$iN(n)$	returns start nodes of all incoming edges of node n
	$at(e)$	returns tags added by edge e
	$rt(e)$	returns tags removed by edge e
	$bt(e)$	returns tags blocked on edge e
	$ot(e)$	returns tags $\notin [at(e), rt(e)]$
	$start(e)$	returns start node of edge e
	$end(e)$	returns end node of edge e
	$phy(e)$	returns physical edge associated with edge e

Table 2: *Variables and Functions*

failures required to disconnect *src* from *dst*.

$$\textbf{Objective:} \qquad \texttt{minimize} \sum_{i \in pEdges} F_i \tag{1}$$

Forwarding constraints. We first discuss the constraints added to represent *traffic forwarding*. The base constraint states that *src* originates the traffic. To disconnect the graph, the next constraint states that the traffic must not reach *dst*.

$$R_{src} = 1 \tag{2}$$

$$R_{dst} = 0 \tag{3}$$

For other nodes, the constraint is that traffic can reach a node n if it gets *propagated* (A_e) on any incoming edge e.

$$\forall n \in oNodes, R_n = \bigvee_{e \in iE(n)} A_e \tag{4}$$

(Logical AND/OR can be expressed as ILP constraints.)
Traffic can propagate through an edge e: if it reaches the start node of e ($start(e)$); if the traffic does not carry a tag that is blocked on e ($\neg B_e$); and if e represents an inter-device edge, the underlying physical link does not fail ($\neg F_{phy(e)}$). This is represented as

$$\forall n \in oNodes, \forall e \in iE(n) :$$
$$A_e = R_{start(e)} \wedge \neg B_e \wedge \neg F_{phy(e)} \tag{5}$$

Tag Constraints. We now add constraints to model route tags. Route tags propagate in route advertisements from processes that add tags to processes that remove tags or block advertisements based on tags. However, the TPG models traffic propagation, which occurs in the opposite direction of route propagation. Hence, instead of modeling the flow of tags from the nodes that add them to the nodes that block/remove them, we model the flow of *blocked tags* from the nodes that block them to the nodes that remove/add them. The base constraints state that each edge that blocks on the tag "forwards" the blocked tag, and each edge that removes the blocked tag does not forward it.

$$\forall e \in edges, \forall t \in bt(e) :$$
$$T_{e,t} = 1 \tag{6}$$

$$\forall e \in edges, \forall t \in rt(e) :$$
$$T_{e,t} = 0 \tag{7}$$

For other edges, we add the constraint that edge e forwards a blocked tag t iff the start node of e ($start(e)$) receives traffic with t.

$$\forall e \in edges, \forall t \in ot(e) :$$
$$T_{e,t} = \bigvee_{i \in iE(start(e))} T_{i,t} \tag{8}$$

Finally, we add the constraint that an edge e carries blocked traffic iff e receives a tag that is added by e.

$$\forall e \in edges :$$
$$B_e = \bigvee_{t \in at(e)} T_{e,t} \tag{9}$$

B.2 Tiramisu Longest Path

We now present the complete ILP for BOUND (§6.2). For ease of reading, we repeat our description of the constraints.

Recall that our objective is to maximize the number of inter device edges ($dEdges$) traversed by traffic (A_i).:

$$\textbf{Objective:} \qquad \texttt{maximize} \sum_{i \in dEdges} A_i \tag{10}$$

Path constraints. To ensure traffic flows on only one path, *src* sends traffic, and *dst* receives traffic, we add the following constraints:

$$\sum_{out \in oE(src)} A_{out} = 1 \tag{11}$$

$$\sum_{in \in iE(dst)} A_{in} = 1 \tag{12}$$

For other nodes, we add the flow conservation property, i.e. the sum of incoming flows is equal to the outgoing flows.

$$\forall n \in oNodes : \sum_{in \in iE(n)} A_{in} = \sum_{out \in oE(n)} A_{out} \tag{13}$$

Traffic constraints. Next, we add constraints on traffic propagation. Traffic will be blocked on edge e if it satisfies the tag (B_e) constraints. This is similar to Eqn 9.

$$\forall e \in edges : A_e \leq \neg B_e \tag{14}$$

C Proofs

We first prove the correctness of *TPVP* (§C.1). This theorem is then used to prove the correctness of our ILPs(§C.2) and *TDFS* (§C.3).

C.1 *TPVP*

In this section, we prove that we correctly model and solve the stable paths problem. Griffin *et al* [15] showed that *an instance of the stable paths problems is a graph together with the permitted paths $\mathcal{P}$ at each node and the ranking functions for each node.* Hence, we first need to prove that we correctly model the permitted paths $\mathcal{P}$ and the ranking function. We will use the following lemmas to prove them. Finally, we will prove that as long as the network converges, *TPVP* finds the exact path taken in the network and hence the solution to the stable path problem.

Lemma 2. *All permitted paths $p \in \mathcal{P}$ that a routing process $v \in V$ may take in the actual network to reach the destination, exists in the TPG ($p \in TPG$).*

Proof: Consider the path $v_i \rightarrow v_{i+1} ... \rightarrow v_n$, where v represents a node in the graph. We will inductively prove that for any path $p \in \mathcal{P}$, a node v_i has a path to the *dst* node in the TPG, if there is an equivalent path that traverses the same routing processes (i.e. the traffic matches the RIB entries of these routing processes) and vlan-interfaces on the real network.

When $i = n$, then the node v_i represents the destination (*dst*) of traffic.

Assume there is a path from node v_{i+1} to v_n and this is modeled correctly. We will now prove that v_i is connected to v_{i+1} in the TPG if v_i can use v_{i+1} to reach the next-hop router (in its path towards the destination node). Node v_i can be one of the following types of nodes

Case (i): node v_i is an OSPF node According to §4.2, node v_{i+1} will be a VLAN-egress node. TPG connects v_i and v_{i+1} if *OSPF* is configured to operate on that *VLAN* interface. OSPF uses its configured interfaces [22] to (a) receive link-state advertisement from its neighbors, and (b) forward/send traffic towards its neighbors. Hence, OSPF will use the VLAN-egress node to forward traffic to its neighbors/next-hop router. Hence case (i) is modeled correctly.

Case (ii): node v_i is an BGP node According to §4.2, node v_{i+1} will be either an OSPF node or an VLAN-egress node. These instances represent BGP using either an OSPF-computed route or a connected subnet to reach the next hop router. Assume v_{i+1} is an *OSPF* node. BGP process is allowed to communicate with its neighboring adjacent process through an OSPF-computed path. Hence these nodes can be connected in the actual network. Assume v_{i+1} is a VLAN-egress node for VLAN interface V. TPG connects them if BGP's next-hop IP address falls within the subnet assigned to VLAN interface V. Similar to case (i), this interface is use to receive advertisements and send traffic to the next-hop router. Hence case (ii) is modeled correctly.

Case (iii): node v_i is a VLAN-ingress node According to §4.2, node v_{i+1} will be either an OSPF/BGP node representing the processes running on the same router, or an egress node for the same VLAN on that router. Assume v_{i+1} is an OSPF/BGP node. In the TPG, all traffic enters a router through a VLAN-ingress node. In the actual network, all incoming traffic looks up the global RIB of the router to move to the next-hop router. A router's global RIB contains entries from all the RIBs of all of its routing processes. The TPG connects the VLAN-ingress node to an OSPF/BGP node if that node is tainted in the RAG. Since only those processes that have a RIB entry for the *dst* are tainted, this is modeled correctly. Assume v_{i+1} is an VLAN-egress node. TPG connects them iff the underlying device is a switch. Since flooding occurs at Layer-2, this is modeled correctly.

Case (iv): node v_i is a VLAN-egress node According to §4.2, node v_{i+1} will be a VLAN-ingress node. TPG connects them if they belong to the same VLAN and their underlying routers are connected in the physical topology. This models basic physical connectivity in the real network. Hence, case (iv) is modeled correctly.

Case (v): node v_i is the *src* node According to §4.2, node v_{i+1} will be either an OSPF node or a BGP node. The *src* node is like any other VLAN-ingress node, with the difference being that the *src* originates the traffic. Hence, using similar arguments as case (iii), case (v) is modeled correctly.

Lemma 3. TPVP *will not choose a path $p \notin \mathcal{P}$ from a routing process $v \in V$ to the destination, that the routing process v cannot choose in the actual network.*

Proof: A TPG is partially based on the physical topology. In the TPG, the only inter-device edge is the edge between the VLAN-egress node of a device and the VLAN-ingress node of its neighboring device. Hence, the TPG will not have any path that does not exist in the actual physical network topology.

In the TPG, *OSPF* and *BGP* nodes are connected to a VLAN-egress node if they processes is configured to operate on that VLAN. Additionally, a VLAN-ingress node is connected to *OSPF* and *BGP* nodes if those processes are tainted in the RAG. Hence, the TPG will correctly connect the interfaces with its associated routing process, and won't have any path due to incorrect route adjacency.

Next, we prove that Tiramisu correctly models prefix/neighbor-based and tag-based filters. The TPG models prefix/neighbor-based filters by removing edges from the node associated with the process that uses those filters. Hence, the TPG will not have any path that is blocked due to prefix/neighbor-based filters in the actual network.

The only path that may exist in the TPG and not in the network is the path blocked by tags. *TPVP* uses routing algebra [16, 25] to model route computation and route selection operations. [16] showed that routing algebra can model and filter routes based on tags. Routing algebra uses "tag" attributes in their edge labels and path signatures. Their

$\oplus$ (operation) function models both addition of tags and blocking routes based on tags. Hence *TPVP* will not choose a path that is blocked by tags.
Hence *TPVP* will not choose a path p that cannot be chosen in the actual network.

Lemma 4. *L3TPG has the same set of layer 3 paths as the actual network and has no path that cannot exist in the actual network.*

Proof: Since advertisements can flow between layer 3 nodes as long as there is at least one layer 2 path that connects them, Tiramisu uses path contraction to replace layer 2 paths with a layer 3 edge.
Assume there is a layer 3 path P^* in the L3TPG that does not exist in the network. Since path contraction replaces subpaths with edges and does not add paths, P^* with (layer 2 nodes inbetween) must exist in the original TPG. I.e. If nodes are connected in the layer-3 graph, then they have to be connected even before path contraction. This contradicts Lemma 3.
Assume there is a layer 3 path $P^{\#}$ in the network that does not exist in the L3TPG. Since path contraction replaces subpaths with edges and does not eliminate paths, $P^{\#}$ with (layer 2 nodes inbetween) must not exist in the original TPG. This contradicts Lemma 2.

In the following lemmas, we operate on the contracted L3TPG.

Lemma 5. *For all routing process $v \in V$, given a set of paths $\mathcal{P}$, $best(\mathcal{P}, v)$ matches the path that v would choose when paths $\mathcal{P}$ are given as choices in the actual network.*

Proof: *TPVP* uses routing algebras to model the ranking behavior of actual protocols (*OSPF* and *BGP*). Routing algebras model route selection/path ranking using a $\preceq$ preference relation over path signatures. This correctly models the route selection algorithm in the underlying network. Hence the path chosen by $best(\mathcal{P}, v)$ for each routing process $v \in V$ matches the path chosen in the actual network.

Lemma 6. *For all routing process $v \in V$, the set of paths $\mathcal{P}$ considered by* TPVP *matches the paths that v can choose from in the actual network.*

Proof: Our networks may run either (i) only *OSPF* processes, (ii) only *BGP* processes, or (iii) both *OSPF* and *BGP* processes. We leverage Lemma 5 to prove the correctness for each scenario.

Case (i): OSPF-only network In the L3TPG, we have only *OSPF* nodes. From Lemma 5, we know that *OSPF* nodes will make the same choice as the actual OSPF processes. Hence case (i) is modeled correctly.

Case (ii): BGP-only network In the L3TPG, we have only *BGP* nodes. From Lemma 5, we know that *BGP* nodes will make the same choice as the actual *BGP* processes. Hence case (ii) is modeled correctly.

Case (iii): OSPF+BGP network In the layer 3 TPG, we have both *BGP* and *OSPF* nodes. Here choices made by *OSPF* or *BGP* nodes may depend on the opposite protocol's choices.

Case (iii-a): both node v_i and v_{i+1} are BGP nodes. This is similar to case (ii). Hence it is modeled correctly.
The next three cases represent instances where *BGP* uses an *OSPF* computed path to reach its next hop.

Case (iii-b): node v_i is a BGP node, node v_{i+1} is an OSPF node, and v_{i+1} is connected to a single BGP node. Since *OSPF* has only one choice, it will select the *BGP* process as its next hop and case (iii-b) is modeled correctly.

Case (iii-c): node v_i is a BGP node, node v_{i+1} is an OSPF node, and v_{i+1} is connected to multiple BGP nodes having equal preference (same cost). Since BGP *processes* are equally preferred, OSPF cost is used to select the best path/route [1]. Case (i) showed that given a set of OSPF-costs, the *OSPF* node will make the right choice. Hence, this case (iii-c) is modeled correctly.

Case (iii-d): node v_i is a BGP node, node v_{i+1} is an OSPF node and v_{i+1} is connected to multiple BGP nodes having different preferences. Since an *OSPF* node cannot make a decision using *BGP* preferences, Tiramisu cannot model this scenario. In the real network, this scenario arises when an iBGP process assigns preferences (lp) or tag-based filters to routes received from its iBGP neighbors.

Case (iii-e): node v_i is a BGP node, node v_{i+1} is an OSPF node and v_{i+1} is connected to one or multiple OSPF nodes. This scenario can only happen if *BGP* and *OSPF* have route redistribution. In this scenario, v_{i+1} makes the choice for intra-domain routing and v_i makes the choice for inter-domain routing. Using similar arguments as case (i) and case (ii), choices made by v_{i+1} and v_i will be correct.

Case (iii-f): node v_i is an OSPF node. Assume node v_{i+1} is an *OSPF* node. This is similar to case (i). Hence it is modeled correctly. Assume node v_{i+1} is a BGP node. This will lead to similar scenarios as case (iii-c) to case (iii-e). Hence, using similar arguments, case (iii-f) is modeled correctly as long as iBGP processes do not assign preferences to routes received from its iBGP neighbors.

Theorem 1. *If the network control plane converges,* TPVP *always finds the exact path taken in the network under any given failure scenario.*

Proof: Lemma 2 and 3 showed that we correctly model permitted paths $\mathcal{P}$. Lemma 5 and 6 showed that we correctly model path selection/ranking function. Now, we need to prove that under convergence, *TPVP* is equivalent to PVP.

The body of the loop code of *TPVP* (line 7 to 11) is equivalent to the *PVP* code that runs at each node [26]. The termination conditions of *PVP* and *TPVP* (line 12 to 13) are also similar. Both of them establish convergence when there are no new messages in the distributed and shared buffers respectively. And finally, both of them are executed after removing edges to represent failures.

The main difference between *PVP* [15, 26] and *TPVP* is the following: In PVP, there is no restriction on the order in which messages are processed by different routers. As long as the network converges and there exists a stable path, *PVP* will find it irrespective of the order in which messages are sent and processed. In *TPVP*, we process and send messages in a fixed round-robin order (line 5). Since any ordering of messages in *PVP* leads to a valid solution, a fixed ordering of messages should also lead to a valid solution. Hence if the network converges, *TPVP* finds the exact path.[4]

C.2 ILP

We will first use two lemmas to prove that *TPVP* and the min-cut ILP consider the same set of paths. Recall that the min-cut ILP can be applied directly only on a TPG with no ACLs; in §6.1, we showed how it deals with ACLs.

Lemma 7. *The min-cut ILP only considers paths computed by TPVP.*

Proof: Assume the min-cut ILP considered a path that was not computed by *TPVP*. This implies the path was ignored by *TPVP* but not the ILP. This is not possible because (i) according to Lemma 2 and 3, *TPVP* correctly models permitted paths, and (ii) the ILP and *TPVP* exclude the same set of paths, i.e. both of them exclude paths based on tags. The ILP constraints on tags (Eqn 6 and 14) ensure that all paths blocked by tags are ignored by the ILP.

Lemma 8. *The min-cut ILP considers all paths computed by TPVP.*

Proof: Assume the min-cut ILP did not consider a path computed by *TPVP*. This implies the path was ignored by the ILP but not by *TPVP*. Similar to Lemma 7, this is not possible because (i) the ILP correctly models permitted paths (Lemma 2 and 3), and (ii) the ILP and *TPVP* exclude the same set of paths.

Theorem 9. *In the absence of ACLs, the min-cut ILP computes the minimal failures to disconnect the src and dst in the TPG.*

Proof: By Lemma 7 and 8, the ILP considers the same set of paths as *TPVP*. Next, we will show that this ILP computes a valid cut. Then, we will show that the cut is minimum.

Assume the ILP does not compute a valid cut. This means there is a path where all its edges A_e are equal to 1. However that also means that by Eqn 4, one of $R_{start(e)}$ is R_{dst} and R_{dst} is equal to 1. This contradicts Eqn 3.

Assume the ILP computes an invalid cut. This means there is still a path where all its edges A_e are equal to 1. Using the same argument, this will again contradict Eqn 4 with Eqn 3.

Assume the cut computed by ILP is not the minimum. This will contradict the objective function (Eqn 1) which minimizes the number of edge failures (removals) to disconnect the graph.

Next, we prove the correctness of the longest-path ILP.

Theorem 10. *The longest-path ILP computes the length of the longest inter-device path between src and dst in the TPG.*

Proof: Note that the longest-path ILP is modeled for the same TPG as the min-cut ILP. Although the ILPs are different, the constraints used to block based on tags are the same. Using similar arguments as min-cut ILP, we can prove the following two lemmas

Lemma 11. *The longest-path ILP only considers paths computed by TPVP.*

Proof: Similar to Lemma 7

Lemma 12. *The longest-path ILP considers all paths computed by TPVP.*

Proof: Similar to Lemma 8

Hence, by Lemma 11 and 12, this ILP also considers the same set of paths as *TPVP*. Next, we will show that this ILP computes a valid path. Then, we will show that this path is the longest.

Assume, the ILP finds an invalid path. This means there is some node which has an outgoing flow without any incoming flow. This contradicts the flow conservation constraint (Eqn 13). Assume, the ILP misses a valid path. This means there is some node which does not have an outgoing flow but has an incoming flow. This also contradicts Eqn 13.

Assume the path computed by ILP is not the longest. This will contradict the objective function (Eqn 10) which maximizes the number of edges traversed to reach the destination.

C.3 TDFS

Theorem 13. TDFS *identifies a src-dst pair as always unreachable iff there never exists a path from src to dst under all possible failures*

Proof: Assume there existed some path in the network that *TDFS* did not consider. This implies the path was ignored by TDFS and not *TPVP*. We will now prove by contradiction why this is not possible.

There are four types of paths that can exist in the network to establish reachability. We will show that *TDFS* will capture each of those paths.

Case (a): assume a path p_1 exists from *src* to *dst* which does not traverse any edge with tags. Assume this path is ignored by *TDFS*. Line 5 of *TDFS* runs *DFS* after removing edges that block on tags. In this case, no edges are removed. Since p_1 connects *src* to *dst*, *DFS* will return true and *TDFS* will say *dst* is reachable from *src*. This contradicts our assumption.

Case (b): assume a path p_2 exists from *src* to *dst* which does not traverse any edge that blocks on a tag. Assume this path is ignored by *TDFS*. Line 5 of *TDFS* runs *DFS* after removing edges that block on tags. Since p_2 connects *src* to *dst* without traversing any of these removed edges, *DFS* will return true and *TDFS* will say *dst* is reachable. This again contradicts our assumption.

Case (c): assume a path p_3 exists from *src* to *dst* that traverses an edge that blocks on tags but does not traverse any edge which adds tags to an advertisement. Assume this path is ignored by *TDFS*. Line 7 of *TDFS* runs *DFS* after (i) establishing that all paths go through edges that block on tags, and, (ii) removing edges that add tags. Because of (i), we know all

[4]Note that Tiramisu assumes network convergence. The routing algebra that models *TPVP* will have all the algebraic properties required for convergence.

p_3 will traverse a tag blocking edge. Since p_3 connects *src* to *dst* (and hence a tag-blocking edge to *dst*) without traversing the removed edges, *DFS* will return true and *TDFS* will say *dst* is reachable. This again contradicts our assumption.

Case (d): assume a path p_4 exists from *src* to *dst* which traverses an edge that blocks on tags, followed by an edge that removes that tag and an edge that adds that tag. Assume *TDFS* ignores this path. Line 9 of *TDFS* runs *DFS* after (i) establishing that all paths go through both edges that block on a tag and add that tag, and (ii) removing edges that remove that tag.

TDFS will return false if *DFS* states that the tag-blocking edge cannot reach the tag-adding edge. This can happen in two scenarios. In the first scenario, the tag-blocking edge could not reach the tag-adding edge even before node removal. In the second scenario, the tag-blocking edge could reach the tag-adding edge, but the removal of the tag-removing edge disconnected them. p_3 already captures the first scenario. And p_4 represents the second scenario. *TDFS* captures both these scenarios. Hence, this again contradicts our assumption.

Assume there exists some path $P^{\#}$ that is considered by *TDFS* as a valid path but does not exist in the network. As mentioned in Lemma 3, the only path that exists in the TPG and not in the network is the path blocked by tags. Hence, $P^{\#}$ must be blocked by tags. However, Line 11 of *TDFS* returns true if all paths that connect *src* to *dst* traverses a tag-blocking edge X, followed by a tag-adding edge Y, and no tag removing edge between X and Y. Hence, *TDFS* correctly identifies and ignores paths based on tags. This again contradicts our assumption.

D Other Policies

Some of the other policies that Tiramisu can verify are listed below:

Always Chain of Waypoints (CW)**.** Similar to waypointing, we remove nodes associated with each waypoint, one at a time. Using *TDFS*, we check if nodes associated with one of the preceding waypoints in the chain can reach nodes associated with one of the following waypoints in the chain.

Equal Bound (EB)**.** This policy checks that all paths from *src* to *dst* are of the same length. The objective of the ILP in §6.2 can be changed to find the shortest path length. If the longest and shortest path length varies, then this policy is violated.

Multipath Consistency (MC)**.** Multipath consistency is violated when traffic is dropped along one path but blocked by an ACL on another. To support multipath in *TPVP*, we change the p_* variable to keep track of multiple most preferred path signatures to reach *dst*. Using *TPVP*, Tiramisu can identify the number of best paths to reach *dst*. We run *TPVP* on graphs with and without removing edges for ACLs. If the number of paths varies with and without ACLs, then the policy is violated.

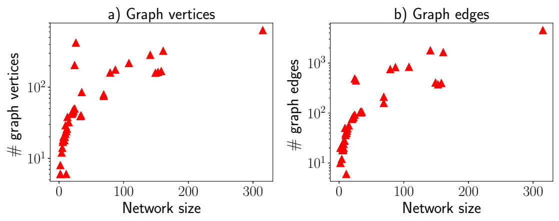


Figure 12: Size of multilayer graphs of all networks

Always no black holes (BH)**.** Black holes occur when traffic gets forwarded to a router that does not have a valid forwarding entry. Blackholes are caused by i) ACLs: routers advertises routes for traffic that is blocked by their ACLs; ii) static routes: the next-hop router of a static route cannot reach *dst*. Tiramisu uses *TDFS* to check these conditions. For (i) Tiramisu first creates the graph without removing edges for ACLs. Let R be the router with a blocking ACL. If *src* can reach router R and R can reach *dst* (using *TDFS*), then traffic will reach router R under some failure, and then get dropped because of the ACL. For (ii) if *src* can reach the router with the static route and the next-hop router *cannot reach dst*, then the traffic gets dropped.

E Protocols/Modifiers in Network

We used university, datacenter, topology zoo, and Rocketfuel configurations in our evaluation §8. Table 3 shows what percentage of networks in these datasets support each network protocol/modifier.

	% of Networks			
Protocols/Modifiers	University	Datacenter	Topology Zoo	Rocketfuel
eBGP	100%	100%	100%	100%
iBGP	100%	0%	100%	100%
OSPF	100%	97%	100%	100%
Static routes	100%	100%	0%	0%
ACLs	100%	100%	0%	0%
Route Filters	100%	97%	100%	0%
Local Prefs	50%	0%	100%	0%
VRF	100%	0%	0%	0%
VLAN	100%	0%	0%	0%
Community	100%	100%	100%	0%

Table 3: Configuration constructs used in networks

Figure 12 characterizes the size of the TPGs generated by Tiramisu for these networks. It shows the number of nodes and edges used to represent the graph. We observe two outliers in both Figure 12a and Figure 12b. These occur for networks *Uni*2 (24 devices) and *Uni*3 (26 devices), from the university dataset. These networks have multiple VRFs and VLANs, and Tiramisu creates multiple nodes (and edges between these nodes) for different VRFs, routing processes and VLAN interfaces. Note also that for the other networks, the number of routing processes per device varies. Hence, the number of nodes and edges do not monotonically increase with network size.

Automated Verification of Customizable Middlebox Properties with Gravel

Kaiyuan Zhang *Danyang Zhuo*[†] *Aditya Akella*[#] *Arvind Krishnamurthy* *Xi Wang*
University of Washington [†] *Duke University* [#] *University of Wisconsin-Madison*

Abstract

Building a formally-verified software middlebox is attractive for network reliability. In this paper, we explore the feasibility of verifying "almost unmodified" software middleboxes. Our key observation is that software middleboxes are already designed and implemented in a modular way (e.g., Click). Further, to achieve high performance, the number of operations each element or module performs is finite and small. These two characteristics place them within reach of automated verification through symbolic execution.

We perform a systematic study to test how many existing Click elements can be automatically verified using symbolic execution. We show that 45% of the elements can be automatically verified and an additional 33% of Click elements can be automatically verified with slight code modifications. To allow automated verification, we build Gravel, a software middlebox verification framework. Gravel allows developers to specify high-level middlebox properties and checks correctness in the implementation without requiring manual proofs. We then use Gravel to specify and verify middlebox-specific properties for several Click-based middleboxes. Our evaluation shows that Gravel avoids bugs that are found in today's middleboxes with minimal code changes and that the code modifications needed for proof automation do not affect middlebox performance.

1 Introduction

Middleboxes (e.g., NATs, firewalls, and load balancers) play a critical role in modern networks. Yet, building functionally correct middleboxes remains challenging. Critical bugs have routinely been found in middlebox implementations. Many of these bugs [8–12] directly lead to system failure or information leaks. Worse still, malformed packets can trigger some of these bugs and expose severe security vulnerabilities.

Given the importance of building functionally correct middleboxes, researchers have turned to formal verification and have made significant progress [14, 40]. Crucially, these efforts tackle real middlebox implementations rather than abstract middlebox models and verify non-trivial program properties. However, just as with using software verification in other areas of computer systems, this can incur a non-trivial amount of proof effort (e.g., 10:1 proof to code ratio in VigNAT [40]). At the same time, the excessive proof effort prevents researchers from exploring verification of high-level middlebox-specific properties (e.g., a middlebox rejects unsolicited external connection). As a consequence, recent verification efforts focus either entirely on low-level code properties (e.g., free of crashes, memory safety) [14] or on proving equivalence to pseudocode-like low-level specifications [39, 40].

In this paper, we ask whether it is possible to make software middlebox verification completely automated with minimal proof effort. In particular, our goal is two-fold. First, we want verification to work on real-world "almost unmodified" middlebox implementations without requiring manual proofs. Second, we want developers to be able to express and verify high-level properties directly translated from RFCs (e.g., RFC5382 [29] for NAT) without writing manual proofs towards each of these properties. To deliver on these goals, we seek to replicate the automated reasoning approach used in some recent verification projects that focus on file systems and OS system calls [30, 34]. Specifically, we would like to use symbolic execution to automatically encode a middlebox implementation and its high-level specification using satisfiability modulo theories (SMT) and then use solvers to verify that the implementation is consistent with the specification.

Our key observation regarding the suitability of this approach is that many existing middleboxes are already designed and implemented in a modular way (e.g., Click [23]) for reusability. As they aim for high performance, the number of operations they perform on each packet is finite and small. Both characteristics place these middleboxes within reach of automated verification through symbolic execution. Thus, one goal of this paper is to identify domain-specific analyses that enable symbolic execution to exploit these characteristics and distill SMT encodings for middlebox implementations.

We begin by studying whether we can use automated verification on existing software middleboxes. We perform a systematic study on all 290 Click elements and 56 Click configurations (≈60K lines of code) in Click's official repository to test whether they are suitable for automated verification. We find that a baseline symbolic executor can derive symbolic expressions for 45% of the elements and 16% of the configurations. We then introduce a set of domain-specific static analyses and code modifications (such as replacing element state by SMT-encoded abstract data types) to enable the symbolic execution of a more substantial fraction of Click elements. These techniques allow us to symbolically execute an additional 33% and 50% of elements and configurations, respectively.

Encouraged by the results of the empirical study, we designed and implemented Gravel, a framework for automated software verification of middleboxes written using Click [23]. Gravel provides developers with programming interfaces to specify high-level trace-based properties in Python. Gravel symbolically executes the LLVM intermediate representation compiled from an element's C++ implementation. Gravel then uses Z3 [38] to verify the correctness of the middlebox without the burden of manual proofs.

We then evaluate Gravel by porting five Click middleboxes: MazuNAT, a load balancer, a stateful firewall, a web proxy, and a learning switch. We verify their correctness against high-level specifications derived from RFCs and other sources. Only 133 out of 1687, 63 out of 1151, 63 out of 1447, 50 out of 953, and 0 out of 594 lines of code need to be modified to make them automatically verifiable. The high-level specification of the middlebox-specific properties can be expressed concisely in Gravel, using only 177, 70, 68, 39, and 91 lines of code. Our evaluation shows that Gravel can avoid bugs similar to those found in existing unverified middleboxes. Finally, we show that the code modifications do not degrade the performance of the ported middleboxes.

2 Encoding Existing Software Middleboxes

To understand the feasibility of applying automated verification to existing software middleboxes, we perform an empirical study of all the 290 Click elements and 56 Click configurations[1] in Click's official repository [23]. In this section, we first explain what is automated verification and then describe ways to enhance the effectiveness of automated verification for middleboxes. Finally, we show that 78% of Click elements and 66% of Click configurations are amenable to automated verification after some limited modifications to the code.

2.1 Automating verification using symbolic execution

A well-established approach to software verification is deductive verification. In this style, a developer generates a collection of proof obligations from the software and its specifications. Proof assistants, such as Coq [7], Isabelle [32], and Dafny [24], are highly expressive, allowing mathematical reasoning in high-order logic. However, the verification process is mostly manual, requiring significant effort from the developer to convey his/her knowledge of why the software is correct to the verification system. For example, when applied to a NAT, VigNAT [40] shows a 10:1 proof-to-code ratio.

Recently, researchers have started exploring the feasibility of automating the verification process through exhaustive symbolic execution, which encodes the middlebox implementation into a symbolic expression that can be checked against a high-level specification. This style of software verification reduces the developers' manual proof effort and has already been used successfully to verify file systems [34] and operating systems [30]. However, this style is more limited than deductive verification, putting restrictions on the programming model. For example, Hyperkernel requires loops in its system call handlers to have compile-time bounds on their iteration counts.

[1]Our empirical study focuses on the Click elements and configurations that process packets in a run-to-completion model.

```
class CntSrc : public Element {
    // omitting constructor and destructor
    Packet *process_packet(Packet *pkt) {
        if (pkt->ip_header->saddr == target_src_)
            cnt_++;
        return pkt;
    }
    IPAddress target_src_;
    uint64_t cnt_;
}
```

Figure 1: A C++ implementation of a simple packet counter.

To see an example of symbolic execution based verification, Figure 1 shows a simple packet counter. This code increments a counter when the source IP address of a packet matches a signature (i.e., `target_src_`). Here we model this `process_packet` function as $f : \mathbb{S} \times \mathbb{P} \mapsto \mathbb{S} \times \mathbb{P}$, where $\mathbb{S}$ is the set of all possible internal states (`target_src_` and `cnt_`) and $\mathbb{P}$ denotes the set of all possible packets. For simplicity, this formulation assumes that at most one outgoing packet is generated for each incoming packet. Symbolically executing this code snippet generates the following symbolic expression:

$$\begin{aligned}
&\forall s,\ s' \in \mathbb{S}, \forall p,\ p' \in \mathbb{P},\ f(s,p) = (s',p') \Rightarrow \\
&\qquad (p' = p) \wedge (s'.target\_src = s.target\_src) \\
&\qquad \wedge (p.saddr = s.target\_src \Rightarrow s'.cnt = s.cnt + 1) \\
&\qquad \wedge (p.saddr \neq s.target\_src \Rightarrow s'.cnt = s.cnt))
\end{aligned}$$

This symbolic expression says that for all possible inputs, outputs and state transitions: (1) the input packet should be the same as the output packet; (2) the `target_src_` should not change; (3) if the packet's source IP address matches `target_src_`, the `cnt_` in the new state should be the `cnt_` in the old state plus 1; (4) if the packet's source IP address does not match `target_src_`, the `cnt_` should not change.

Symbolic execution alone is not enough for automated verification; it only ensures that we can automatically generate the above expression. To ensure automated verification, when the developer verifies the above expression against a specification using an off-the-shelf theorem solver (such as Z3 [38]), the solver needs to be able to solve it efficiently.

A program is suitable for **automated verification** if:

1. Symbolic execution of the program halts and yields a symbolic expression.
2. The resulting symbolic expression is restricted to an effectively decidable fragment of first-order logic.

Condition 1 means the program has to halt on every possible input. Condition 2 depends on which fragment of first-order logic a solver can solve efficiently. This fragment changes as solver technologies improve over time. Empirically, we know that if we can restrict the symbolic expression to only bit vectors and equality with uninterpreted functions, a solver can tackle the expression efficiently [30].

2.2 Baseline effectiveness of symbolic execution

We now study the feasibility of automating verification by examining middleboxes written using Click. We perform an empirical study on both Click elements and Click configurations, which are datagraphs formed by composing Click elements. This study allows us to measure both the fraction of Click code and the fraction of automatically verifiable Click programs. To perform this empirical analysis, we implement a baseline symbolic executor to analyze whether an element or a configuration satisfies the conditions mentioned above. Since elements are C++ classes, the symbolic executor first analyzes all the member fields to determine whether the state could be encoded with SMT (Condition 2). It then performs symbolic execution over the compiled LLVM byte code[2] of the element to check if Condition 1 is met. However, since both conditions are undecidable, we choose to use the following two conservative criteria. (In fact, we describe in the subsequent section how we augment our baseline symbolic executor with domain-specific extensions.)

Absence of pointers in element state. When the symbolic executor analyzes each of an element's members, it checks whether the element state can be expressed solely by bit vectors and uninterpreted functions. Though one could use bit vectors to encode the entire memory into a symbolic state, it would be difficult to efficiently solve expressions containing such a symbolic state due to the sheer size of the search space. Therefore, we choose a conservative criterion, the absence of pointers in element states, as it is easy to see that elements without pointers always have bounded state. Each element in Click can only have a finite number of member variables, and each non-pointer variable can only consume a finite amount of memory. Thus, the state space of a Click element without pointers can always be expressed by constant size bit vectors. Of course, such criteria introduce false negatives, for example, using pointers to access a bounded data structure (e.g., fixed-size array).

Absence of loops and recursions. To determine whether Click elements' execution is bounded (Condition 1), the symbolic executor invokes the packet processing code using a symbolic element state and a symbolic packet content. The symbolic executor detects potential unbounded execution by searching for loops and recursive function calls and only performs execution on those elements that do not contain them. The symbolic executor performs the check by comparing each jump/call target with the history of executed instructions.

Table 1 shows the results of running this baseline symbolic executor. We found that 130 of the existing Click elements (45%) are suitable for automated verification. Among the ones that failed our test, 143 elements failed because of pointers, and 78 elements failed because of unbounded execution. 61 of the elements have both pointers and unbounded execution. A Click configuration is amenable to automated verification if and only if all the Click elements in the configuration can be automatically verified. Among the 56 configurations in the official Click repository, only 9 out of the 56 Click configurations (16%) are suitable for automated verification.

[2]We chose to use LLVM byte code rather than C++ abstract syntax tree as the former makes it easier to reason about the control flow by eliminating C++ related complexities (e.g., function overloading and interface dispatching).

2.3 Enhancing symbolic execution

We now augment our baseline executor with additional techniques that aid symbolic execution. We also examine the impact of performing a small number of code modifications to make the middleboxes amenable to automated verification. Some of the techniques described below are broadly applicable but are likely more effective for middlebox programs that operate on packet data with well-defined protocol specifications. The remaining techniques are domain-specific analyses that are suitable only for packet processing code.

Code unrolling. When detecting a backward jump, the symbolic executor unrolls the loop and executes its loop body. The executor keeps count of how many times it executes the backward jump instruction and raises an error if the number goes beyond a pre-defined threshold. This technique is useful when the source code has loops with a static number of iterations or loops whose iteration count is a small symbolic value, as would be the case for code that processes protocol fields of known size.

Pointer analysis to detect immutable pointers and static arrays. In general, we can classify the use of pointers into three categories: pointers to singleton objects, pointers corresponding to arrays, and pointers used to build recursive data structures. These use cases introduce two distinct challenges in the symbolic execution of Click code with pointers. First, the symbolic executor needs to determine whether two pointers point to overlapping memory regions and update the symbolic state of elements correctly irrespective of which pointer is used for the update. Second, when pointers are used to implement recursive data structures, such as linked list or tree, the data structure access often involves loops whose iteration counts depend on the symbolic state of the elements. Our symbolic executor first identifies how pointers are used and then uses the appropriate technique for symbolic execution.

We first use an analysis pass to identify immutable pointers by checking which of the pointer fields in a Click element remain unmodified after allocation. At the same time, we determine which of the other program variables serve as possible aliases for a given pointer field. Further, for pointers pointing to an array of data items, the symbolic executor also performs a static bounds check on accesses performed using the pointers to ensure that all accesses are within allocated regions. By doing so, the symbolic executor can prove an invariant that accesses performed using the array pointer do not touch other memory regions.

After performing these analyses, the symbolic executor limits itself to handling accesses through immutable and unaliased pointers that refer to either singleton objects or arrays. For each pointer referring to a singleton object, the executor associates a corresponding symbolic value. For each pointer referring to an array of data, the symbolic executor uses an uninterpreted function in SMT to represent the contents of the array. The symbolic executor uses uninterpreted functions that map array index (64-bit integer) to bytes (8-bit integer) to model the content of the array. We choose to use this offset-to-bytes mapping as the unified representation for both array and packet content since reinterpreting a sequence of bytes in memory as a different type is a common practice in packet processing (e.g., parsing packet header, endianness conversion). We record updates to the array as a sequence of (possibly symbolic) index/value pairs. Since the functions are "uninterpreted", they model all possible values of the array data. Compared with bit vectors, representing states with uninterpreted functions makes symbolic execution scale to larger state size [6, 34].

Our symbolic executor does not handle pointers that are used to build a recursive data structure, such as a linked list, except in the case of certain abstract data types for which we are able to provide SMT encodings (as discussed next).

SMT encodings of commonly used abstract data types. Our next technique avoids the symbolic execution of the data structure implementation by hiding the implementation under a well-defined data structure interface. This technique allows us to integrate implementations that may contain unbounded loops or recursive data structures into our analysis. When performing the symbolic execution, we can simply provide an encoding in SMT for common data structures, such as `HashSet`. Note that not all data structures can have their interfaces encoded in SMT. The key challenge here is to prevent the explosion of the state space; the size of the encoding should not depend on the actual size of the data structure. We managed to encode three commonly used data structures in Click, `Vector`, `HashSet`, `HashMap`, into SMT. (See Appendix A.)

Replace element state with abstract data types. With the SMT encoding of common data types, another technique we could apply is to modify the element implementation by replacing its states with the data types mentioned above. This process requires the developer to inspect how the packet processing code uses a specific element state. If all the accesses performed on the state can be modeled using the interface of a data type with SMT encoding, we could replace the state with the SMT-encoded counterpart and run the symbolic execution on the modified implementation instead.

Consider the `CheckIPAddress` element (Figure 2). This element serves as a source IP packet filter. Before our proposed modifications, `CheckIPAddress` stores a list of bad IP addresses (`bad_src_`). A packet is dropped if the source IP address of the packet is listed in the bad IP address list. In

```
class CheckIPAddress : public Element {
    // omitting constructor and destructor
    Packet *process_packet(Packet *pkt) {
        auto saddr = pkt->ip_header->saddr;
-       for (size_t i = 0; i < num_bad_src_; i++)
-           if (bad_src_[i] == saddr)
+       if (bad_src_.find(saddr) != bad_src_.end())
                return NULL;
        return pkt;
    }
-    IPAddress *bad_src_;
-    size_t num_bad_src_;
+   HashSet<IPAddress> bad_src_;
}
```

Figure 2: Modification of `CheckIPAddress`'s implementation to remove the usage of pointers and loops.

this element, `bad_src_` and `num_bad_src_` together represents a fixed size array containing the bad IP addresses. To check whether the source IP address of a received packet matches any address in the array, `CheckIPAddress` uses a "for" loop to go through this array. `CheckIPAddress` is not suitable for automated verification: (1) The size of the array that `bad_src_` is pointing to is not known by the symbolic executor; thus, it may flag out-of-bound memory access. (2) If the executor tries to unroll the loop, it faces a path explosion problem as the number of iterations in the loop can be large.

To make this element meet the conditions for automated verification, we can modify its implementation, as shown in Figure 2. This change is based on the observation that the way `bad_src_` and `num_bad_src_` are used complies with the `HashSet` interface. The change replaces the pointer-size pair `bad_src_` and `num_bad_src_` with a `HashSet`. Besides that, the "for" loop to check whether the source IP is in the bad IP address list is also replaced with a `find` method call. The code changes remove both the use of pointers and unbounded loops. Since the semantics of `HashSet` and its `find` interface is modeled with SMT, we can symbolically execute the element.

Concretization of control flow structures. Middleboxes perform packet classification based on the value of specific fields in the packet header. Packet classification is implemented using finite-state machines, and it is often optimized by statically compiling the classification rules into a state machine model that is stored in memory. When processing an incoming packet, the classifier performs state transitions using the rules until the state machine reaches one of the end states. If the values of the state transition table are abstract, then the classification process would appear to be unbounded.

We address this issue and enable the symbolic execution of the classification tasks. We load the Click configuration containing concrete classification rules and run the state machine creation code of the classification element. We then ingest the raw bytes representing the transition rules into symbols with concrete values. We use symbolic execution to verify that the

Technique	# Elements	# Conf.
Unmodified	130 (45 %)	9 (16 %)
Code unrolling	138 (48 %)	9 (16 %)
Fix-sized array detection	185 (63 %)	9 (16 %)
SMT-encoded abstract datatype	218 (75 %)	13 (23 %)
Replacing with abstract datatype	222 (77 %)	15 (27 %)
Concretization	226 (78 %)	37 (66 %)

Table 1: Number of Click elements and configuration that can be symbolically executed.

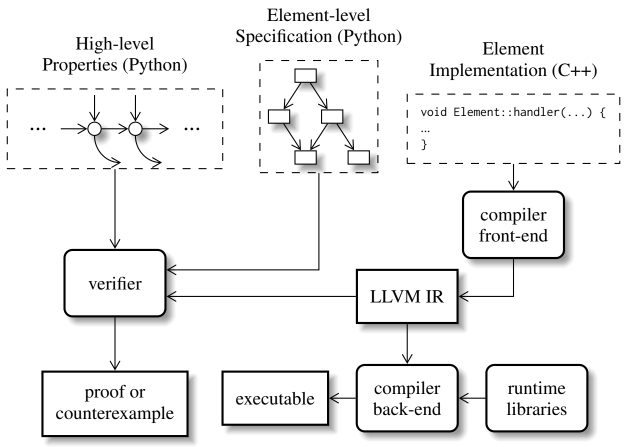
contents of the memory region representing the state transition table remain unchanged during program execution. We then symbolically execute the packet classification code but replace the symbolic transition rules with the concrete values identified in the first step. The executor can thus process the packet classification code within a statically bounded number of steps.

2.4 Overall effectiveness of symbolic execution

We now repeat our analysis of Click elements and configurations after enhancing our symbolic executor with these additional techniques. Table 1 shows the result. Our techniques improve the fraction of elements that can be symbolically executed from 45% to 78%. The fraction of Click configurations that are suitable for automated verification improves from 16% to 66%.

Our symbolic executor cannot handle 22% of the Click elements. Among the 64 unsupported elements, 19 of them could not be symbolically executed because there are loops that traverse the payload of the packets (e.g., `AES` element for encryption). Another 26 elements use customized data structures that contain pointers that can not be modeled with SMT. One such example is `LookupIP6Route` element that uses a match table with longest prefix matching as opposed to a traditional exact match hash table. 11 elements contain loops whose number of iterations is based on the current (symbolic) element state. For example, the `AggregateFilter` element, which aggregates incoming packets according to their header values, has to loop over a queue to determine which aggregation group a packet should belong to. 8 elements have pointer accesses that are deeply coupled with the rest of the code that replacing with abstract data types is not feasible. For example, `IP6NDSolicitor` uses a set of linked lists to handle the response messages of the neighbor discovery protocol.

Three approaches can potentially improve Gravel's ability to verify more Click elements automatically. The first approach is to model more data structures using SMT. Currently, Gravel only supports `HashMap`, `HashSet`, and `Vector`. The second approach is to allow developers to write annotations to rule out part of the implementation that is not relevant to the specification. For example, if the developers only want to prove that the `AES` element does not change the TCP header of the packet, the symbolic executor can skip over the loop that traverses the packet payload. The third approach is to use an interactive theorem prover (e.g., Coq [7], Dafny [24]) to verify the correctness of element-level implementations. These interactive theorem provers can verify higher-order logic than what SMT can verify. For example, more sophisticated data structures such as priority queues or an LRU cache could be more easily verified with the help of an interactive prover.



High-level Properties (Python)
Element-level Specification (Python)
Element Implementation (C++)
void Element::handler(...) {
...
}
compiler front-end
verifier
LLVM IR
proof or counterexample
executable
compiler back-end
runtime libraries

Figure 3: Development Flow of Gravel. Top three boxes denote inputs from middlebox developers; rounded boxes denote compilers and verifiers of Gravel; rectangular boxes denote intermediate and final outputs.

3 The Gravel Framework

Gravel is a framework for specifying and verifying Click [23]-based software middleboxes. It aims to verify high-level properties, such as a load balancer's connection persistency, against a low-level C++ implementation. Gravel uses symbolic execution to translate the C++ implementation into a symbolic expression automatically, and it uses the techniques described in the previous section to enhance the effectiveness of symbolic execution. In this section, we describe how Gravel allows developers to specify the desired high-level properties using Python code and a domain-specific library containing verification primitives. In Section 4, we describe how we check whether the symbolic expression derived from the implementation provides the desired properties.

3.1 Overview

Figure 3 shows the workflow of Gravel. Gravel expects three inputs from middlebox developers:

1. Click configuration, which is a directed graph of elements.
2. A set of high-level middlebox specifications.
3. Element-level specifications for all Click elements used in the configuration.[3]

[3]Gravel provides specifications for commonly used elements.

Like building a normal Click middlebox, Gravel first takes as input a directed graph of Click elements. In Click, a middlebox is decomposed into smaller packet processing "elements". Each element keeps private state that is accessible only to itself and has a set of handlers for events such as incoming packet or timer events. Elements can also have many input and output ports through which elements can be connected with others and transfer packets. The directed graph from a Click configuration connects Click elements together to form the dataplane for packet processing. The topology of the directed graph remains unchanged during the execution of the middlebox.

Gravel then requires a formalization of the high-level middlebox properties. To check properties automatically with an SMT solver, they need to be expressed using first-order logic. In Gravel, properties are formalized as predicates over a trace of events. Gravel includes a Python library for developers to specify middlebox-specific properties.

Gravel also requires a specification for each Click element. The element-level specification describes each element's private state and packet processing behavior. The element-level specification provides a simplified description of an element's behavior and omits low-level details such as performance optimizations. Gravel again provides a Python library for developers to write element-level specifications.

With these three inputs, Gravel verifies the correctness of the middlebox in two steps. First, Gravel checks whether a Click configuration composed using Click elements satisfies the desired high-level properties of the middlebox. A high-level property is expressed as a symbolic trace of the middlebox's behavior (in Python). Gravel verifies the high-level property by symbolically executing the datagraph of elements using element-level specifications (in Python). Then, Gravel verifies that the low-level C++ implementation of each element has equivalent behavior as the element-level specification. Gravel compiles the low-level C++ implementation into LLVM intermediate representation (LLVM IR) and then symbolically executes the LLVM IR to obtain a symbolic expression of the element. Gravel then checks whether the element-level specification holds in the element's symbolic expression. If there is any bug in the Click configuration or the implementation of the elements, Gravel outputs a counterexample that contains element states and an incoming packet that makes the middlebox violate its specification.

3.2 A Sample Application: ToyLB

The rest of this section describes the Gravel framework in the context of a simple running example corresponding to a Layer-3 load balancer, ToyLB. ToyLB receives packets on its incoming interface and forwards them to a pool of servers in a round-robin fashion. It steers traffic by rewriting the destination IP on the packet. ToyLB resembles popular Layer-3 load balancer designs used by large cloud providers [16, 19].

The ToyLB middlebox is decomposed into five elements, as shown in Figure 4. When there is an incoming packet, it first goes through two header-checking elements, `CheckIPHeader` and `CheckTCPHeader`. These two elements act like filters and discard any packet that is not a TCP packet. Then, the `FlowTable` element checks whether the packet belongs to a TCP flow that has been seen by ToyLB earlier. If so, `FlowTable` rewrites the packet with the corresponding backend server's IP address stored in the FlowTable and sends the packet to the destination server. Otherwise, the `FlowTable` consults a `RoundRobinSwitch` scheduler element to decide which backend server should the new connection bind to. After the `RoundRobinSwitch` decides which backend server to forward the packet to, `RoundRobinSwitch` notifies the `FlowTable` of the decision. The `FlowTable` stores the decision into its internal state and also rewrites the destination address of the packet into the destination server. For further simplicity, low-level functionalities such as ARP lookup are omitted in ToyLB.

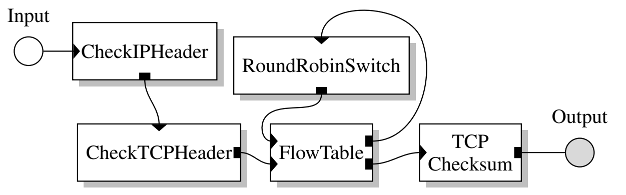


Figure 4: Breakdown of ToyLB's functionalities into packet-processing elements.

We next describe how Gravel can be used to model high-level specifications of middleboxes such as ToyLB and then outline how the element-level properties are specified. Later, in §4, we show how Gravel performs verification.

3.3 High-level Specifications

Gravel models the execution of a middlebox as a state machine. State transitions can occur in response to external events such as incoming packets or passage of time. The time event can be used to implement garbage collection for middlebox states. For each state transition, the middlebox may also send packets out.

Gravel provides a specification programming interface, embedded in Python, for developers to specify high-level properties. Developers can use the interface to describe middlebox behavior over a symbolic event sequence. (See Appendix A.)

Packets in Gravel's high-level specification are expressed using key-value map abstraction, where the keys are the name of header fields and values are the content of the fields. This abstraction makes the specification concise and hides the implementation details that are less related to high-level properties (e.g., the position of IP addresses in the packet header).

Gravel provides three kinds of core interfaces (see Appendix A) in its high-level specification: (1) a set of `sym_*` functions that allow developers to create symbolic representations of different types of states such as IP address, packet, or middlebox state; (2) middlebox's event-handling functions, like handle_packet(state, pkt), handle_time(state, timestamp), that takes as input the current state of the middlebox and the incoming packet/time event, and returns an (optional) output packet and the resulting middlebox state after a state transition; and (3) the verify(formula) function call that first encodes the given logical formula in SMT and invokes the SMT solver to check if formula is always true. Besides that, Gravel also provides some helper functions for developers to encode high-level middlebox properties.

To make this concrete, we next describe how to encode two high-level properties of ToyLB using this specification programming interface. We describe how to encode two load balancer properties: (1) liveness and (2) connection persistence. We first consider the liveness guarantee.

Property 1 (ToyLB liveness)**.** For every TCP packet received, ToyLB always produces an encapsulated packet.

In Gravel, this can be specified as:

```
def toylb_liveness():
    # create symbolic packet and symbolic ToyLB state
    p, s0 = sym_pkt(), sym_state()
    # get the output packet after processing packet p
    o, s1 = handle_packet(s0, p)
    verify(Implies(is_tcp(p), Not(is_none(o))))
```

In this liveness formulation, we first construct a symbolic packet p and the symbolic state of the middlebox s0. Then, we let the middlebox with state s0 process the packet p by invoking the handle_packet function. After that, the state of the middlebox changes to s1, and the output from the middlebox is o. If o is None, the middlebox has not generated an outgoing packet. This high-level specification says that, if the incoming packet is a TCP packet, the middlebox has an outgoing packet.

Note that the formulation of liveness property is abstract, given that it does not say anything about the states of the middlebox. We don't even formulate the set of data structures used by ToyLB. This brevity is indeed the benefit of using high-level specifications. These formulations are concise and are directly related to the desired middlebox properties.

Now, we move to a more complex load balancer property—connection persistency. This property is crucial to a load balancer as it ensures that packets from the same TCP connection are always forwarded to the same backend server.

Property 2 (ToyLB persistency)**.** If ToyLB forwards a TCP packet to a backend b at time t, subsequent packets of the same TCP connection received by ToyLB before time $t + WINDOW$, where $WINDOW$ is a pre-defined constant, will also be forwarded to b.

Formulation of Property 2 is more complex than the liveness property because it requires a forwarding requirement (i.e., the forwarding of packets of a certain TCP connection to b) to hold over all possible event sequences between time t and time $t + WINDOW$. This complexity means that we cannot formulate connection persistency with traces containing only a single event, but rather, we need to use induction to verify that the property holds on event traces of unbounded length.

Gravel allows us to specify Property 2 as an inductive invariant. First, we formulate the forwarding condition that should be held during the time window. The steer_to function defined below determines whether a packet received at time t will be forwarded to the backend server with address dst_ip. The code snippet first lets the middlebox handle a time event with timestamp t, followed by the handling of pkt. We ascertain whether the packet is forwarded to dst_ip by checking that the output from the packet processing is not None and that the resulting packet's destination address is dst_ip.

```
def steer_to(state, pkt, dst_ip, t):
    o0, s_n = handle_time(state, t)
    o1, s_n2 = handle_packet(s_n, pkt)
    return And(Not(is_none(o1)),
               o1.ip4.dst == dst_ip,
               payload_eq(o1, pkt))
```

Then, for the base case of induction, we specify that once ToyLB forwards a packet of a particular TCP connection to a backend, subsequent packets from the same connection received within a period *WINDOW* will be forwarded to the same backend. Similar to the formulation of the liveness property, the following code snippet first creates two symbolic packets and a symbolic middlebox state, then invokes handle_packet to obtain the output packet as well as the new state after packet processing. After that, the code requires the verifier to prove that if p0 is forwarded to dst_ip, then a packet, p1, in the same connection received any time before the expiration time ddl is also forwarded to dst_ip, assuming that the middlebox state hasn't changed from state s1.

```
def base_case():
    p0, p1, s0 = sym_pkt(), sym_pkt(), sym_state()
    o, s1 = handle_packet(s0, p0)
    dst_ip, t0 = sym_ip(), s0.curr_time()
    t = sym_time()
    ddl = t0 + WINDOW
    verify(Implies(And(Not(is_none(o)),
                       o.ip4.dst == dst_ip,
                       from_same_flow(p0, p1)),
              ForAll([t], Implies(t <= ddl,
                      steer_to(s1, p1, dst_ip, t)))))
```

In addition to the base case invariant, the specification includes two inductive cases showing that processing an additional event (e.g., a packet from a different connection or time event) does not change the forwarding behavior. The two inductive cases specify that the invariant steer_to(...) holds on the middlebox states when processing packets or time events if the timestamp is before the expiration time.

```
def step_packet():
    dst_ip, p0, p1 = sym_ip(), sym_pkt(), sym_pkt()
    s0, t0, p_other = sym_state(), sym_time(), sym_pkt()
    o, s1 = handle_packet(s0, p_other)
    verify(Implies(And(steer_to(s0, p0, dst_ip, t0),
                       from_same_flow(p0, p1)),
                   steer_to(s1, p1, dst_ip, t0)))
```

```
def step_time():
    dst_ip, p0, p1 = sym_ip(), sym_pkt(), sym_pkt()
    s0, t0, t1 = sym_state(), sym_time(), sym_time()
    _, s1 = handle_time(s0, t1)
    verify(Implies(And(steer_to(s0, p0, dst_ip, t0),
                       t1 < t0, from_same_flow(p0, p1)),
                   steer_to(s1, p1, dst_ip, t0)))
```

3.4 Element-level Specifications

Verifying high-level specifications directly from low-level C++ implementations is hard because of the gap in their semantics. Similar to all the seminal work [20, 30, 34] in software verification, we break down the verification process using refinement. Gravel requires the developer to give specifications of each element. As long as the element-level specifications capture the behavior of their corresponding elements' implementation, we can simply use the element-level specifications to prove the high-level specifications. Compared to deductive verification, this incurs a lower verification effort because element-level specifications are short (§5). Element-level specifications can be reused across different middleboxes. The element-level specification in Gravel consists of two parts: the definition of abstract states that will be used by the element during execution, and a set of event handling behaviors in response to incoming packets and time events.

Element states. Specification of a Gravel element starts with a declaration of the state associated with the element. To ensure efficient encoding with SMT, Gravel requires the state to be bounded. More specifically, elements' state in Gravel may contain: (1) fixed-size variables including bit vectors; (2) maps from one finite set to another (e.g., a map from IP address space to 64-bit integer). For example, in ToyLB, the state of FlowTable is defined as:

```
class FlowTable(Element):
    num_in_ports = 2
    num_out_ports = 2

    decisions = Map([AddrT, PortT, AddrT, PortT], AddrT)
    timestamps = Map([AddrT, PortT, AddrT, PortT], TimeT)
    curr_time = TimeT
    ...
```

This part of element-level specifications defines three components of FlowTable's state:

- decisions maps from a TCP connection to a backend server address. FlowTable identifies a TCP connection by the tuple of source and destination addresses and port numbers. This map is used to store the results from the Selector element.
- timestamps stores the latest times at which packets were received for each TCP flow stored in decision.
- curr_time stores the current time.

Here the types such as AddrT and TimeT are pre-defined integers of different bit widths. Besides the state, the code also informs Gravel as to how many input/output ports the FlowTable element has through num_in_ports/num_out_ports.

```
def flowtable_process_packet(s, p, in_port):
    flow = p.ip4.saddr, p.tcp.sport, \
           p.ip4.daddr, p.tcp.dport
    # the case when flowtable has record of the flow
    known_flow = And(
        # packet is received from the network
        in_port == IN_TCP_FILTER,
        # flowtable has record of the flow
        flow in s.decisions)
    # construct the encapsulated packet
    fwd_pkt = p.copy()
    fwd_pkt.ip4.dst = s.decisions[flow]
    # update the timestamp of the flow with current time
    after_fwd = s.copy()
    after_fwd.timestamps[flow] = s.curr_time
    known_flow_action =
        Action(known_flow,
               {PORT_TO_EXT: fwd_pkt}, after_fwd)
```

Figure 5: Example of an element-level action.

Event handlers. Gravel requires each element to have a handler function for packets received from its input ports. This packet handler needs to be specified in the element-level specification. The specification of the packet handler describes the operations the element performs when handling packets. Besides that, an optional time event handler can also be specified. In Gravel, the two event handlers are defined as functions with the following signatures:

```
flowtable_process_packet(state, pkt, in_port) → actions
flowtable_process_time(state, timestamp) → actions
```

The return value of each event handler (actions) is a list of *condition-action pairs*. Each entry in the list describes the action an element should take under certain conditions. In the python code, developers can write:

```
Action(cond, { port_i : pkt_i }, new_state)
```

to denote an action that sends pkt_i to output port port_i while also updating the element state to new_state. This action will be taken when condition cond holds. To make it concrete, let us consider the packet handler of FlowTable. Upon receiving a packet, FlowTable does one of the followings:

- If the packet is from the CheckTCPHeader element, and the decisions map contains a record for the connection, FlowTable rewrites the destination address and sends the packet to TCP Checksum element, as shown in Figure 5.
- If the FlowTable does not have a record for a packet, the packet is sent to RoundRobinSwitch element.
- If the packet is sent from RoundRobinSwitch, FlowTable records the destination decided by RoundRobinSwitch and forwards the packet to TCP Checksum.

Similarly, FlowTable's behavior in response to time changes is also specified as *condition-actions*:

```
def flowtable_process_time(self, s, time):
    new = s.copy()
    # update the "curr_time" state
    new.curr_time = time
    # records with older timestamps should expire
    def should_expire(k, v):
        return And(s.timestamps.has_key(k),
                   time >= WINDOW + s.timestamps[k])

    new.decisions = new.decisions.filter(should_expire)
    new.timestamps = new.timestamps.filter(should_expire)
    return Action(True, {}, new)
```

When `FlowTable` is notified of a time change, it updates its `curr_time` to the given time value. Gravel offers a `filter` interface for its map object, which takes a predicate, `should_expire`, and deletes all the entries that satisfy the predicate. `FlowTable` uses this to remove all the records that were inactive for a period longer than a constant `WINDOW` value. ToyLB's complete element-level specifications are in Appendix B.

Summary: Overall, we presented an example of (1) how to specify high-level trace-based middlebox properties, and (2) how to write element-level specifications to make verification modular. We provide a framework for developers to articulate complex trace-based properties. These high-level properties are implementation-independent. Element-level specifications decouple verification problem into two orthogonal problems: that element-level specifications conform to the high-level properties and that elements' implementations comply with their element-level specifications.

4 Verifier Implementation

Gravel proves the middlebox properties with two theorems:

THEOREM 1 (Graph Composition). *The element-level specifications, when composed using the given Click configuration, complies with the high-level specification of the middlebox.*

THEOREM 2 (Element Refinement). *The C++ implementation of Click is a refinement of that element's specification. That is, every possible state transition and packet processing action of the C++ implementation must have an equivalent counterpart in the element-level specification.*

Theorem 1 verifies that the composition of element-level specifications meets the requirement in the high-level specifications. Theorem 2 verifies that Click's C++ implementation of each element meets its element-level specification.

4.1 Graph Composition

Gravel verifies the Graph Composition theorem (Theorem 1) in two steps. First, Gravel symbolically executes the event sequence specified in high-level specifications. Second, Gravel checks whether the high-level specifications hold on the resulting state and outgoing packets from symbolic execution.

Gravel performs symbolic execution on the directed graph. Before the symbolic execution, Gravel creates a symbolic state of the entire middlebox, which is a composition of the symbolic states of all elements in the middlebox. Remember that the high-level specification describes required middlebox behavior on an event sequence. The goal of the symbolic execution is to reproduce the sequence symbolically. For example, if the high-level specification contains an incoming packet, Gravel generates a symbolic packet at the source element of the directed graph. This symbolic packet, when processed by the first element of the graph, can trigger handlers of other downstream elements, which are symbolically executed as well. If the element-level specification contains a branch (e.g., depending on the packet header, a packet can be forward to one of the two downstream elements), Gravel performs symbolic execution in a breadth-first search manner.

After performing symbolic execution for each event type, Gravel records the updated state of each element as well as the packet produced by each output element. Gravel provides this information as the return value of the `handle_*` functions in the high-level specification. Gravel then invokes the functions defined in the high-level specification. Once the `verify` function is invoked, Gravel encodes the high-level specifications into SMT form and uses a solver to see if they always hold.

Loops in the graph. Gravel allows the directed graph of elements to contain loops in order to support bi-directional communications between elements, such as `FlowTable` and `RoundRobinSwitch` in ToyLB (§3). However, loops may introduce non-halting execution when we symbolically execute the datagraph. Gravel addresses this issue by setting a limit on the number of elements traversed by the symbolic executor. When the symbolic execution hits this limit, Gravel raises an alert and fails the verification. For example, in ToyLB, the `FlowTable` is hit at most twice: when `FlowTable` cannot find a record for a certain packet, the packet is sent to `RoundRobinSwitch`, which will later send the packet back to `FlowTable`; upon receiving packets from `RoundRobinSwitch`, `FlowTable` records the selected backend server into its own records and does not send the packet back to `RoundRobinSwitch`. Thus, the maximum number of elements traversed during the symbolic execution is 6, and developers can safely set 6 as the limit for ToyLB.

The graph composition verifier is implemented with 1981 lines of Python. It exposes a similar set of interfaces as Click configuration language so that developers could port existing code into the verifier. The verifier uses the Python binding of Z3 to generate symbolic packets and element states.

4.2 Element Refinement

Gravel verifies the Element Refinement theorem (Theorem 2) in two steps. First, a symbolic expression of the element is generated for each event handler's compiled LLVM intermediate representation. Second, Gravel checks if the element's specification holds on the symbolic expression.

Before performing the symbolic execution, Gravel first uses the LLVM library to extract the memory layout of the C++ class of the element, along with the types of each of its member variables. The verifier can later use this information to determine which field is accessed when it encounters memory access in LLVM bytecode. As mentioned in §2.3, to bound the symbolic execution step and state size, abstract data structures are executed by using their abstract SMT model instead of actual code. A complete list of the data structures and interfaces replaced is given in Appendix A.

For packet content access and modification, Gravel's symbolic executor is compatible with Click's `Packet` interface. In the LLVM bytecode, packet content accesses are compiled into memory operations over a memory buffer. To establish the relation between packet header fields and memory offsets, Gravel needs to extract the symbolic header field value for each output packet after the symbolic execution. Gravel first computes offsets for each header field. Note that these offsets are also symbolic values as they depend on the content of other packet fields. After that, Gravel extracts the value of each header field from the memory buffer of the packet. Each extracted value is then encoded into an SMT formula and compared against fields from the abstract packet using an SMT solver. Gravel concludes that the packet and the memory buffer are equivalent when values of all fields are equivalent.

At the end of symbolic execution, the verifier gets a list of ending states, along with the packets sent out at each output port and the path conditions under which it can be reached. For each entry in the list, Gravel uses Z3 to find an equivalent counterpart in the element specification. If such a counterpart exists for all entries, the refinement of the element is proved.

Gravel's element refinement verifier is implemented in C++ using the LLVM library. The verifier invokes LLVM library's IR parser and reader to load and symbolically execute the compiled LLVM bytecode of each Click element. Besides the SMT encoding of all LLVM instructions used in the compiled Click elements, the verifier also has the SMT encoding of the abstract data types as described in §2. The refinement verifier and the symbolic executor consists of 10396 lines of C++.

4.3 Trusted Computing Base

The trusted computing base (TCB) of Gravel includes the verifier (used for proving Theorem 1 and Theorem 2), the high-level specifications, the tools it depends on (i.e., the Python interpreter, the LLVM compiler framework, and the Z3 solver), and Click runtime. Note that the specification of each element is not trusted.

5 Evaluation

This section aims to answer the following questions:

- How much effort is needed to port existing Click applications? Can Gravel scale to verify the Click applications?
- Can Gravel's verification framework prevent bugs?
- How much run-time overhead does the code modification introduce to middleboxes in order for them to be automatically verifiable by Gravel?

5.1 Case Studies

To evaluate whether Gravel can work for existing Click applications, we port five Click applications to Gravel. For each application, we choose a set of high-level middlebox-specific properties either by formalizing them directly or extracting them from existing RFCs. We use Gravel to verify that these properties hold. Gravel also verifies the low-level properties, such as memory safety and bounded execution.

		LOC	Verif. Time (s)	LOC changed
MazuNAT	Impl	1687	–	133
	Spec (element)	443	64.60	–
	Spec (high-level)	177	3.78	–
Firewall	Impl	1151	–	63
	Spec (element)	73	32.30	–
	Spec (high-level)	70	0.67	–
Load Balancer	Impl	1447	–	63
	Spec (element)	101	10.87	–
	Spec (high-level)	68	1.48	–
Proxy	Impl	953	–	50
	Spec (element)	92	30.63	–
	Spec (high-level)	39	0.72	–
Switch	Impl	594	–	0
	Spec (element)	131	27.73	–
	Spec (high-level)	91	1.61	–

Table 2: Development effort and verification time of using Gravel on five Click-based middleboxes.

MazuNAT: MazuNAT is a NAT that has been used by Mazu Networks. MazuNAT consists of 33 Click elements. (See Appendix C.) MazuNAT forwards traffic between two network address spaces, the internal network, and the external network. It mainly performs two types of packet rewriting:

1. For a packet whose destination address is the NAT, the NAT rewrites its destination IP address and port with the corresponding endpoint in the internal network.
2. For a packet going from the internal to the external network, NAT assigns an externally visible source IP address and port to the connection. The NAT also needs to keep track of assigned addresses and ports to guarantee persistent address rewriting for packets in the same connection.

One common property we verified for all five middleboxes is that the middlebox does not change the packets' payload:

PROPERTY 3 (Payload Preservation)**.** For any packet that is processed by the middlebox, the middlebox never modifies the payload of the packet.

For NAT-specific properties, we verified that MazuNAT meets the requirements proposed in RFC5382 [29].[4] These requirements are proposed to make NATs transparent to applications running behind them [17].

PROPERTY 4 (Endpoint-Independent Mapping)**.** For packets p_1 and p_2 from the same internal IP, port $(X : x)$, where

[4]We omit the set of requirements related to ICMP becuase MazuNAT does not support ICMP.

- p_1 targets external endpoint $(Y_1 : y_1)$ and gets its source address and port translated to $(X'_1 : x'_1)$
- p_2 targets external endpoint $(Y_2 : y_2)$ and gets its source address and port translated to $(X'_2 : x'_2)$

the NAT should guarantee that $(X'_1 : x'_1) = (X'_2 : x'_2)$.

PROPERTY 5 (Endpoint-Independent Filtering)**.** Consider external endpoints $(Y_1 : y_1)$ and $(Y_2 : y_2)$. If the NAT allows connections from $(Y_1 : y_1)$, then it should also allow connections from $(Y_2 : y_2)$ to pass through.

PROPERTY 6 (Hairpinning)**.** If the NAT currently maps internal address and port $(X_1 : x_1)$ to $(X'_1 : x'_1)$, a packet p originated from the internal network whose destination is $(X'_1 : x'_1)$ should be forwarded to the internal endpoint $(X_1 : x_1)$. Furthermore, the NAT also needs to create an address mapping for p's source address and rewrite its source address accordingly.

These properties are essential to ensure the transparency of the NAT and are required for TCP hole punching in peer-to-peer communications.

We also prove that the MazuNAT preserves the address mapping for a constant amount of time:

PROPERTY 7 (Connection Memorization)**.** If at time t, the NAT forwards a packet from a certain connection c, then for all states s' reachable before time $t + THRESHOLD$, where *THRESHOLD* is a predefined constant value, packets in c are still forwarded to the same destination.

Property 7 guarantees that the NAT can translate the address of all packets from a TCP connection consistently. The constant *THRESHOLD* defines a time window where the TCP connection should be memorized by the NAT. The NAT has the freedom to recycle the resources used for storing connection information after the time window expires.

Load Balancer: Besides the round-robin load balancer mentioned in §3, we also verified a load balancer using Maglev's hashing algorithm [16]. Its element graph looks exactly the same as in Figure 4. The only difference is that the `RoundRobinSwitch` element is replaced by a hashing element that uses consistent hashing. The load balancer steers packets by rewriting the destination IP address.

We verified connection persistency for both of the load balancers. The goal of connection persistency is to make load-balancing transparent to the clients.

PROPERTY 8 (Load Balance Persistence)**.** For all packets p_1 and p_2 from connection c, if the load balancer steers p_1 to a backend server, then the load balancer steers p_2 to the same backend server before c is closed.

Stateful Firewall: The stateful firewall is adapted from the firewall example in the Click paper [23]. Besides performing static traffic filtering, it also keeps track of connection states between the internal network and the external network. The firewall updates connection states when processing TCP control packets (e.g., *SYN*, *RST*, and *FIN* packets), and removes records for connections that are finished or disconnected.

We prove that the stateful firewall can prevent packets from unsolicited connections [28]. Also, the firewall should garbage collect finished connections.

PROPERTY 9 (Firewall Blocks Unsolicited Connection)**.** For any connection c, no packet in c from the external network is allowed until a *SYN* packet has been sent out for c.

PROPERTY 10 (Firewall Garbage-collects Records)**.** For any connection c, no packet in c from the external network is allowed after the firewall sees a *FIN* or *RST* packet for c.

Web Proxy: The Web proxy transparently forwards all web requests to a dedicated proxy server. When the middlebox receives a packet, it first identifies if it is a web request by checking the TCP destination port. For web request packets, the proxy rewrites the packet header to redirect them to the proxy server. The proxy also memorizes the sender of the web request to forward the reply messages back to the sender.

We prove that the web proxy middlebox forwards packets in both directions.

PROPERTY 11 (Web Proxy Bi-directional)**.** For a web request packet p with 5-tuple `(SA,SP,DA,DP,PROTO)`, if the middlebox forwards p to the proxy server and rewrites the 5-tuple to `(SA',SP',DA',DP',PROTO)`, then a packet from the reply flow with 5-tuple `(DA',DP',SA',SP',PROTO)` should be forwarded back to the sender.

Learning Switch: The Learning switch implements the basic functionality of forwarding Ethernet frames and MAC learning. The switch learns how to send to an Ethernet address A by watching which interface packets with source Ethernet address A arrives. If the switch has not learned how to send to an Ethernet address, it broadcasts the packet to all its interfaces.

We prove the following properties about the switch.

PROPERTY 12 (Forwarding Non-interference)**.** For any Ethernet address A, the behavior of how the switch forwards packets targeting A is not be affected by packets whose source Ethernet address is not A.

PROPERTY 13 (Broadcasting until Learnt)**.** For any address A, if the switch broadcasts packets targeting A, it keeps broadcasting until a packet from A is received by the switch.

5.2 Verification Cost

To understand the cost of middlebox verification on Gravel, we evaluate the amount of development effort and the verification time. Table 2 shows the result.

Development effort. We find that porting existing Click applications to Gravel requires little effort and that writing specifications with Gravel are also easy. We only modified 133 lines of code in MazuNAT to make it compatible with Gravel. The firewall and load balancer required only 63 lines

Middlebox	Bug ID	Description	Can be prevented?	Why/Why not?
Load Balancer	bug #12	Packet corruption	✓	high-level specification
	bug #11	Counter value underflow	✓	element refinement
	bug #10	Hash function not balanced	✗	not formalized in specification
	bug #6	throughput not balanced	✗	not formalized in specification
Firewall	bug #822	Counter value underflow	✓	element refinement
	bug #691	segfault by uninitialized pointer	✓	element refinement
	bug #1085	Malformed configuration leading crash	✗	Gravel assumes correct init
NAT	bug #658	Invalid packet can bypass NAT	✓	element refinement
	bug #227	Stale entries may not expire	✓	high-level specification
	bug #148	Infinite loop	✓	element refinement

Table 3: Bugs from real-world software middleboxes.

of code modifications. Our proxy required 50 lines of code to be changed, and the switch requires no modification. Most of the required code changes come from the `IPRewriter` element. We had to remove the priority queue that is used for flow expiration and instead use a linear scan to expire old mappings. Other code changes include removing pointers to other elements in `FTPPortMapper`, replacing `ARPTable` in `ARPQuerier` with hashmaps, and the change of `CheckIPHeader` mentioned in §2. The specifications are concise. The high-level specification is below 200 lines of code and the element-level specifications are less than 450 lines of code for all five middleboxes. The associated developer effort is also small. For the web proxy and learning switch, it took less than one person-day for both the high-level properties and the element specifications. The load balancer and the stateful firewall each required a full day's effort in order to port them to Gravel and verify their correctness. The most complicated middlebox in our case study, MazuNAT, took about 5 person-days to port and specify. Five elements (`Classifier`, `IPClassifier`, `IPRewriter`, `CheckIPHeader`, and `EtherEncap`) are reused across these middleboxes, and thus we reuse their element-level specifications.

Verification time. With Gravel's two-step verification process, Gravel's verifier can efficiently prove that the middlebox applications provide the desired properties. Most of the verification time is spent on proving the equivalence of the C++ implementation of each element and its element-level specification. Verification of the high-level specifications from the element-level specifications took less than 4 seconds for the different applications. Overall, even for MazuNAT, the overall verification time is just over a minute.

5.3 Bug prevention

When verifying MazuNAT with Gravel, we found that the original MazuNAT implementation did not possess the endpoint independent mapping property (Property 4). MazuNAT uses a 5-tuple as the key to memorize rewritten flows. This means that when MazuNAT forwards a packet coming from the external network, the packet's source IP address and source port affects the forwarding behavior, violating Property 4. To fix this, we changed the `IPRewriter` element to use only a part of the 5-tuple when memorizing flows.

To evaluate the effectiveness of Gravel at a broader scope, we manually analyze bugs from several open-source middlebox implementations. We wanted to understand whether these bugs can happen if the middlebox is built using Gravel. We examine bug trackers of software middleboxes with similar functionalities as those in our case studies (i.e., NAT, load balancer, firewall) and search the CVE list for related vulnerabilities. We inspect bug reports from the NAT and firewall of the netfilter project [31], and the Balance load balancer [3]. Since the netfilter project contains components other than the NAT and the firewall, we use the bug tracker's search functionality to find bugs relevant only to its NAT and firewall components. We inspect the most recent 10 bugs for all three kinds of middleboxes and list the result in Table 3.

Of the 30 bugs we inspected, we exclude 10 bugs for features that are not supported in our middlebox implementations, 3 bugs related to documentation issues, 5 bugs on command-line interface, and 2 bugs on performance.

From the remaining 10 bugs, Gravel's verifier is able to catch 7 of them. Among these bugs, *Bug #12* in the load balancer and *bug #227* in the NAT can be captured by the verification of the high-level specification as they lead to the violation of Property 3 and Property 7 respectively. Other bugs involving integer underflow or invalid memory access can be captured by the C verifier. Note that there are still three bugs Gravel cannot capture, such as incorrect initialization of the system and properties that are not in our high-level specifications (e.g., unbalanced hashing).

5.4 Run-time Performance

To examine the run-time overhead introduced by the code modifications we made, we compare the performance of the middleboxes before and after the code modifications. We run these Click middleboxes on DPDK [13].

Our testbed consists of two machines each with Intel Xeon E5-2680 (12 physical cores, 2.5 GHz), running Linux (v4.4) and has a 40 Gbps Mellanox ConnectX-3 NIC. The two machines are directly connected via a 40 Gbps link. We run the

		Throughput (Gbps)	Latency (μs)
NAT	Unverified	37.39 ($\pm$ 0.03)	14.43 ($\pm$ 0.19)
	Gravel	37.41 ($\pm$ 0.04)	15.14 ($\pm$ 0.22)
LB	Unverified	37.38 ($\pm$ 0.04)	14.82 ($\pm$ 0.23)
	Gravel	37.37 ($\pm$ 0.04)	14.86 ($\pm$ 0.20)
Firewall	Unverified	37.37 ($\pm$ 0.05)	15.21 ($\pm$ 0.20)
	Gravel	37.38 ($\pm$ 0.04)	15.11 ($\pm$ 0.24)
Proxy	Unverified	37.36 ($\pm$ 0.05)	14.54 ($\pm$ 0.19)
	Gravel	37.35 ($\pm$ 0.06)	14.35 ($\pm$ 0.18)
Switch	Unverified	37.36 ($\pm$ 0.05)	15.02 ($\pm$ 0.19)
	Gravel	37.39 ($\pm$ 0.07)	14.96 ($\pm$ 0.29)

Table 4: Performance of verified middleboxes, compared to their unmodified counterparts.

middlebox application with DPDK on one machine and use the other machine as both the client and the server.

The code modification to make these Click applications compatible with Gravel has minimal run-time overhead. We measure the throughput of 5 concurrent TCP connections using *iperf*, and use *NPtcp* for measuring latency (round trip time of a 200-byte TCP message). Table 4 shows the results. The code modifications introduce negligible overheads in terms of throughput and latency.

6 Related Work

Middlebox verification. Verifying the correctness of middleboxes is not a new idea. Software dataplane verification [14] uses symbolic execution to catch low-level programming errors in existing Click elements [23]. Our work is also based on Click, but we target high-level middlebox-specific properties, such as load balancer's connection persistency. In addition, we show that 78% of existing Click elements are amenable for automated verification with slight code modifications. VigNAT [40] proves a NAT with a low-level pseudocode specification. Vigor [39] generalizes VigNAT to a broader class of middleboxes and verifies the underlying OS network stack and the packet-processing framework. We believe it is non-trivial to extend VigNAT and Vigor to specify and verify the set of high-level trace-based NAT properties (e.g., hairpinning, endpoint-independence) Gravel can verify.

We note though that specifying the correctness of programs is a fundamentally hard problem. Gravel chooses to let developers specify high-level specifications on a symbolic trace of packets. We find specifications using Gravel's specification interface to be more abstract than psuedo-code like NAT specification in VigNAT [40]. However, even with Gravel, writing specifications is still hard. For example, specifying the connection persistency property for ToyLB requires the usage of induction (§3.3). Empirically, we find that a trace-based specification is flexible enough to express the correctness of middleboxes in the RFCs we examined.

Network verification. In the broader scope of network verification, most existing work [1, 2, 4, 15, 21, 22, 26, 27, 33, 37] targets verifying network-wide objectives (e.g., no routing loop) assuming an abstract network operation model. Gravel, along with other middlebox verification work [14, 39, 40], aims to verify the low-level C++ implementation of a single middlebox's implementation. As switches become programmable [5], researchers have built tools to debug [36], verify [18, 25] P4 programs. Similar to Gravel, this line of work relies heavily on symbolic execution. Our work targets "almost unmodified" middleboxes written in C++.

Currently, Gravel only supports verification of middleboxes implemented with Click. However, since our key observation on Click middleboxes, that the number of operations performed processing each packet is finite and small, may also hold on non-Click middleboxes, we believe that Gravel's verification techniques can also be applied on other middleboxes. For example, the eXpress Data Path (XDP) in the Linux kernel also constrains the packet processing code to be loop-free. It also only allows a limited set of data structures for maintaining global states. These properties make it seem plausible that one could apply Gravel's verification techniques to it.

SMT-based automated verification. Automated software verification using symbolic execution has recently become popular. This technique has been used to successfully verify file systems [34], operating systems [30], and information flow control systems [35]. However, this technique usually requires a complete re-implementation of the target application because of the restricted programming model. We conduct a systematic study on (§2) whether unmodified Click-based software middleboxes can be automatically verified.

7 Conclusion

Verifying middlebox implementations has long been an attractive approach to obtain network reliability. We explore the feasibility of verifying "almost unmodified" software middleboxes. Our empirical study on existing Click-based middleboxes shows that existing Click-based middleboxes, with small modifications, are suitable for automated verification using symbolic execution. Based on this, we have designed and implemented a software middlebox verification framework, Gravel. Gravel allows verifying high-level trace-based middlebox properties of "almost unmodified" Click applications. We ported five Click applications to Gravel. Our evaluation shows that Gravel can avoid bugs found in existing middleboxes with small proof effort. Our evaluation also shows that the modifications required for automated verification incur negligible performance overheads. Gravel's source code is available at https://github.com/Kaiyuan-Zhang/Gravel-public.

Acknowledgments

We thank our shepherd Bryan Parno and the anonymous reviewers for their helpful feedback on the paper. This work was partially supported by NSF (CNS-1714508) and Futurewei.

References

[1] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic Foundations for Networks. In *POPL* (2014).

[2] ARASHLOO, M. T., KORAL, Y., GREENBERG, M., REXFORD, J., AND WALKER, D. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *SIGCOMM* (2016).

[3] Balance, Inlab Networks. https://www.inlab.net/balance/.

[4] BECKETT, R., GUPTA, A., MAHAJAN, R., AND WALKER, D. A General Approach to Network Configuration Verification. In *SIGCOMM* (2017).

[5] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[6] CADAR, C., DUNBAR, D., ENGLER, D. R., ET AL. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI* (2008).

[7] COQ DEVELOPMENT TEAM. *The Coq Proof Assistant Reference Manual, Version 8.5pl2*. INRIA, July 2016. http://coq.inria.fr/distrib/current/refman/.

[8] CVE-2013-1138. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1138.

[9] CVE-2014-3817. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3817.

[10] CVE-2014-9715. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9715.

[11] CVE-2015-6271. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6271.

[12] CVE-2017-7928. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7928.

[13] Data Plane Development Kit. https://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html.

[14] DOBRESCU, M., AND ARGYRAKI, K. Software Dataplane Verification. In *NSDI* (2014).

[15] DUMITRESCU, D., STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Dataplane Equivalence and Its Applications. In *NSDI* (2019).

[16] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI* (2016).

[17] FORD, B., SRISURESH, P., AND KEGEL, D. Peer-to-peer communication across network address translators. In *USENIX ATC* (2005).

[18] FREIRE, L., NEVES, M., LEAL, L., LEVCHENKO, K., SCHAEFFER-FILHO, A., AND BARCELLOS, M. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *SOSR* (2018).

[19] GANDHI, R., LIU, H. H., HU, Y. C., LU, G., PADHYE, J., YUAN, L., AND ZHANG, M. Duet: Cloud Scale Load Balancing with Hardware and Software. In *SIGCOMM* (2014).

[20] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving Practical Distributed Systems Correct. In *SOSP* (2015).

[21] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header Space Analysis: Static Checking for Networks. In *NSDI* (2012).

[22] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI* (2013).

[23] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *TOCS* (2000).

[24] LEINO, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Berlin, Heidelberg, 2010), LPAR'10, Springer-Verlag, pp. 348–370.

[25] LIU, J., HALLAHAN, W., SCHLESINGER, C., SHARIF, M., LEE, J., SOULÉ, R., WANG, H., CAŞCAVAL, C., MCKEOWN, N., AND FOSTER, N. p4v: Practical Verification for Programmable Data Planes. In *SIGCOMM* (2018).

[26] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking Beliefs in Dynamic Networks. In *NSDI* (2015).

[27] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the Data Plane with Anteater. In *SIGCOMM* (2011).

[28] MOSHREF, M., BHARGAVA, A., GUPTA, A., YU, M., AND GOVINDAN, R. Flow-level State Transition as a New Switch Primitive for SDN. In *HotSDN* (2014).

[29] NAT Behavioral Requirements for TCP. Available from IETF https://tools.ietf.org/html/rfc5382.

[30] NELSON, L., SIGURBJARNARSON, H., ZHANG, K., JOHNSON, D., BORNHOLT, J., TORLAK, E., AND WANG, X. Hyperkernel: Push-Button Verification of an OS Kernel. In *SOSP* (2017).

[31] The netfilter.org Project. https://www.netfilter.org.

[32] NIPKOW, T., PAULSON, L. C., AND WENZEL, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Feb. 2016.

[33] PANDA, A., LAHAV, O., ARGYRAKI, K. J., SAGIV, M., AND SHENKER, S. Verifying Reachability in Networks with Mutable Datapaths. In *NSDI* (2017).

[34] SIGURBJARNARSON, H., BORNHOLT, J., TORLAK, E., AND WANG, X. Push-button Verification of File Systems via Crash Refinement. In *OSDI* (2016).

[35] SIGURBJARNARSON, H., NELSON, L., CASTRO-KARNEY, B., BORNHOLT, J., TORLAK, E., AND WANG, X. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *OSDI* (2018).

[36] STOENESCU, R., DUMITRESCU, D., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Debugging P4 programs with Vera. In *SIGCOMM* (2018).

[37] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. SymNet: Scalable Symbolic Execution for Modern Networks. In *SIGCOMM* (2016).

[38] The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[39] ZAOSTROVNYKH, A., PIRELLI, S., IYER, R., RIZZO, M., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. Verifying Software Network Functions with No Verifcation Expertise. In *SOSP* (2019).

[40] ZAOSTROVNYKH, A., PIRELLI, S., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. A Formally Verified NAT. In *SIGCOMM* (2017).

A Gravel Programming Interface

A.1 High-level Specification Interface

Table 5 gives a list of the interfaces Gravel offers to the developers. The core interfaces of Gravel includes:

- Functions that generates symbolic value (bitvectors) of different sizes (the `sym_*` API).
- Functions that performs graph composition and returns the result of packet or event processing (`handle_*`)
- The `verify` function which informs Gravel's verifier the verification task to perform.

Besides the core interfaces, Gravel also provides a set of helper functions to ease the formalization effort. These functions include functions that access header fields and functions that checks whether two packets are from the same TCP flow. Table 5 also lists some examples of helper functions.

A.2 Modeling Abstract Data Structure

As discussed in §4, Gravel masks the actual C++ implementation of several data structures and replace them with an SMT encoding during the symbolic execution in order to generate SMT expressions that could be efficiently reasoned about by SMT solvers. Table 6 lists all the interfaces that Gravel's symbolic executor masks during the verification process. This section gives more details on how Gravel generates SMT encoding for these data structure interfaces in a way that the resulting formular can be effciently solved.

Unlike bounded data such as the content of a network packet or an integer field in element state, which can be encoded as a symbolic byte sequence using the bitvector theory of SMT, these data structures have a large state space. This means that encoding them with bitvectors does not results in practically solvable expression. For example, the state of a `HashMap<IPAddress, IPAddress>` could grow up to $2^{64}-1$ bytes. This sheer size makes it infeasible to be encoded using bitvectors.

Gravel's symbolic executor choose to use a different approach and represents data structures as a set of uninterpreted functions. In the aforementioned `HashMap` example, Gravel represents the map as two functions:

$$f_{contain} : \{0,1\}^{32} \mapsto \{\perp, \top\}$$
$$f_{value} : \{0,1\}^{32} \mapsto \{0,1\}^{32}$$

$f_{contain}$ maps from the key space $\{0,1\}^{32}$ to boolean space and represents whether certain key is present in the `HashMap`. Similarly, f_{value} represents the mapping between hashmap keys and the corresponding values.

Each of the data structure interfaces is also modeled by Gravel as operations performed on uninterpreted functions. For the `find(K k)` interface of `HashMap`, Gravel first gets the symbolic value representing whether the key is in the map

Function name	Description
`Core Interfaces:`	
`sym_*()` $\rightarrow$ `SymValT`	Create a symbolic value of corresponding type
`handle_packet(s, pkt, in_port)` $\rightarrow$ `o1, ⋯, on, ns`	Handle the packet and returns the outputs and new state
`handle_time(s, timestamp)` $\rightarrow$ `o1, ⋯, on, ns`	Handle time event, return value is same as `handle_packet`
`verify(formula)`	Encode given formula and verify that a formula always holds
`Helper Functions:`	
`is_none(output)` $\rightarrow$ `Bool`	Check if an output is `None`
`payload_eq(p1, p2)` $\rightarrow$ `Bool`	Determine if two packets have the same payload
`from_same_flow(p1, p2)` $\rightarrow$ `Bool`	Determine if two packets are from the same TCP connection
`is_tcp(pkt)` $\rightarrow$ `Bool`	Check if a packet is TCP packet

Table 5: Gravel's specification programming interface.

by computing $f_{contain}(k)$. Based on the result, Gravel takes different actions:

$$\text{If } f_{contain}(k) = \top, \texttt{find(k)} = f_{value}(k)$$
$$\text{If } f_{contain}(k) = \bot, \texttt{find(k)} = \bot$$

In the actual implementation, $\bot$ is represented as `HashMap::end()`.

The `intert(K k, V v)` interface performs update on the content of the `HashMap`. In Gravel, this is modeled as creating a new set of uninterpreted functions, $f'_{contain}$ and f'_{value} such that:

$$\begin{aligned}&\forall k' \in \{0,1\}^{32} \cdot \\ &\quad f'_{contain}(k') = (f_{contain}(k') \vee (k = k')) \\ &\wedge (k \neq k') \Rightarrow f'_{value}(k') = f_{value}(k') \\ &\wedge f'_{value}(k) = v\end{aligned}$$

Similarly, `erase(K k)` replaces $f_{contain}$ with a new function $f'_{contain}$ such that:

$$\forall k' \in \{0,1\}^{32} \cdot f'_{contain}(k') = f_{contain}(k') \wedge (k \neq k')$$

Besides modeling interfaces from existing Click code base, Gravel also adds a set of iteration interfaces that corresponds to commonly used data structure traverse paradigms. These interfaces could be used to abstract away loops in the Click implementation and making more elements feasible for automated verification.

Gravel currently provides two interfaces for `HashMap`, `map` and `filter`. for `map` interface, Gravel takes as parameter a function g and replace f_{value} with a function f'_{value} where:

$$\forall k \in \{0,1\}^{32} \cdot f'_{value}(k) = g(k, f_{value})$$

Similarly, `filter` takes a predicate p and create a function $f'_{contain}$ such that:

$$\forall k \in \{0,1\}^{32} \cdot f'_{contain}(k) = p(k, f_{value})$$

The modeling of interfaces of `Vector` and `HashSet` are similar to the modeling of `HashMap` mentioned above. The main difference are that `HashSet` only uses $f_{contain}$ function, where as `Vector` uses a symbolic integer to denote the size of the vector and does not have a $f_{contain}$ function.

B ToyLB's Element-level Specification

This section gives a detailed description of the element-level specification of ToyLB. As mentioned in §3, element-level specification in Gravel is given as a list of "condition-action" pairs. In Gravel, developers write python functions that generates the list of possible actions for an element. For example, The `CheckIPHeader` element only forwards packets that are both IP packets and are not from a known "bad" address:

```
def checkipheader_process_packet(s, p, in_port):
    is_bad_src = p.ip.src in s.bad_src
    return [Action(And(p.ether.ether_type == 0x0800,
                       Not(is_bad_src)),
                   {0: p},
                   s)]
```

Remember that the `Action` is used to create a *condition-action* entry, which denotes an action that the element takes under certain condition (§3).

Similarly, `CheckTCPHeader` filters all packets that are not TCP packets.

```
def checktcpheader_process_packet(s, p, in_port):
    return [Action(p.ip.proto == 6,
                   {0: p},
                   s)]
```

`RoundRobinSwitch` not only performs address rewriting for incoming packets, it also updates packet header fields and its own state:

```
def roundrobinswitch_process_packet(s, p, in_port):
    ns, np = s.copy(), p.copy()
    dst_ip = s.addr_map[s.cnt]
    ns.cnt = (s.cnt + 1) % s.num_backend
    np.ip4.dst = dst_ip
    return [Action(True, {0: np}, ns)]
```

The FlowTable element have a more complex specification as it takes one of three actions based on both the content of the incoming packet and its own state:

```
def flowtable_process_packet(s, p, in_port):
    flow = p.ip4.saddr, p.tcp.sport, \
           p.ip4.daddr, p.tcp.dport
    # the case when flowtable has record of the flow
    known_flow = And(
        # packet is received from the network
        in_port == IN_TCP_FILTER,
        # flowtable has record of the flow
        flow in s.decisions)
    # construct the encapsulated packet
    fwd_pkt = p.copy()
    fwd_pkt.ip4.dst = s.decisions[flow]
    # update the timestamp of the flow with current time
    after_fwd = s.copy()
    after_fwd.timestamps[flow] = s.curr_time

    known_flow_action =
        Action(known_flow,
               {PORT_TO_EXT: fwd_pkt}, after_fwd)

    # the case when flowtable does not know the flow
    consult_sched = And(
        in_port == INPORT_NET,
        Not(flow in s.decisions))
    unknown_flow_action =
        Action(consult_sched, {PORT_TO_SCHED: p}, s)

    # packet from the Scheduler
    register_new_flow = in_port == IN_SCHED
    # extract the new_flow
    new_flow = p.inner_ip.saddr, p.tcp.sport, \
               p.inner_ip.daddr, p.tcp.dport
    # add the record of the new_flow to FlowTable
    after_register = s.copy()
    after_register.decisions[new_flow] = p.ip4.daddr
    after_register.timestamps[new_flow] = s.curr_time
    register_action =
        Action(register_new_flow, {PORT_TO_EXT: p},
               after_register)

    return [known_flow_action,
            unknown_flow_action,
            register_action]
```

C Verifying Properties of MazuNAT

The MazuNAT middlebox is the most complicated application Gravel verifies in the case study (§5.1). Figure 6 shows the directed graph of Click elements extracted from its configuration file.

The three properties of MazuNAT proved by Gravel are extracted from RFC [29]. They are important to provide transparency guarantees for application running inside the network. Here we give the formalization of them in Gravel using Gravel's Python interface.

Payload Preservation (Property 3). The specification of Property 3 simply says that the payload of any packet forwarded by the middlebox remains the same. Note that this is a general property that can be verified on multiple middleboxes.

```
def test_payload_unchanged(self):
    p, s = sym_pkt(), sym_state()
    for source in sources:
        ps, _ = handle_packet(s, source, p)
        for sink in sinks:
            verify(Implies(Not(ps[sink].is_empty()),
                           ps[sink].payload == p.payload))
```

Endpointer Independent Mapping (Property 4. For Property 4, the specification starts by creating two symbolic packets, p1 and p2. It then invoke the process_packet on both packets (using the same symbolic state s). After that, it asks the verifier to check if the rewritten packets sending to the external network have the same source address.

```
def to_external(p, s):
    return p.ip.dst != s.public_ip

def same_src(p1, p2):
    return And(is_tcp_or_udp(p1), is_tcp_or_udp(p2),
               p1.ip.src == p2.ip.src,
               src_port(p1) == src_port(p2))

def test_ep_independent_map(self):
    p1, p2, s = sym_pkt(), sym_pkt(), sym_state()

    out1, _ = handle_packet(s, 'from_intern', p1)
    out2, _ = handle_packet(s, 'from_intern', p2)
    o1 = out1['to_extern']
    o2 = out2['to_extern']
    verify(Implies(And(to_external(p1, s),
                       to_external(p2, s),
                       same_src(p1, p2)),
                   same_src(o1, o2)))
```

Endpoint Independent Filtering (Property 5). The high-level specification of Property 5 starts with creating symbolic packet p1 and symbolic state s. Then it creates a new packet p2 by replace only the source address and port with fresh symbolic values. After that the specification uses process_packet to get the resulting packets from processing p1 and p2. Finally, we ask the verifier to check whether the resulting packets (o1 and o2 in the code snippet below) are sent to the same destination.

```
def test_ep_independent_filter(self):
    p1, s = sym_pkt(), sym_state()
    ps1, _ = handle_packet(s, 'from_extern', p1)
    p2 = p1.copy()
    p2.ip.src = sym_ip()
    p2.tcp.src = sym_port()
    p2.udp.src = sym_port()
    ps2, _ = handle_packet(s, 'from_extern', p2)
    for sink in sinks:
        o1 = ps1[sink]
        o2 = ps2[sink]
        verify(Implies(Not(o1.is_empty()),
                       And(Not(o2.is_empty()),
                           o1.ip.dst == o2.ip.dst,
                           dst_port(o1) == dst_port(o2))))
```

Hairpinning (Property 6). As shown below, rather than inspecting the state of elements in MazuNAT to determine whether a address mapping is established. Gravel uses the packet forwarding behavior as the indicator. The specification says that if a packet p1 from external network is forwarded

Function name	Description
`Vector<T>:`	
`const T& get(unsigned int)`	Get value by index
`void set(unsigned int i, T v)`	Set i-th value of vector to `v`
`void map(void(*)(T) f)`	Apply function `f` for all value in vector
`HashMap<K, V>:`	
`V &find(K k)`	Lookup by key `k`
`void insert(K k, V v)`	Insert key-value pair `k, v` into the hashmap
`void erase(K k)`	Delete key `k` from the hashmap
`void map(void(*)(K k, V v) f`	Apply function `f` to all key-value pair in hashmap
`void filter(bool(*)(K k, V v) p)`	Filter key-value pairs in the hashmap with predicate `p`
`HashSet<T>:`	
`T &find(T v)`	Check if `v` is present in hashset
`void insert(T v)`	Insert `v` into the hashset
`void erase(T v)`	Delete `v` from the hashset
`void filter(bool(*)(T v) p)`	Filter with predicate `p`

Table 6: Data structure interfaces supported by Gravel.

to internal network. any packet p2 with the same destination address and port received from internal network is also forwarded to the same destination in the internal network.

```
def test_hairpinning(self):
    p1, p2, s = sym_pkt(), sym_pkt(), sym_state()
    out1, _ = handle_packet(s, 'from_extern', p1)
    out2, _ = handle_packet(s, 'from_intern', p2)
    o1 = out1['to_intern']
    o2 = out2['to_intern']
    verify(Implies(And(p1.ip.dst == p2.ip.dst,
                       p1.ip.proto == p2.ip.proto,
                       dst_port(p1) == dst_port(p2),
                       o1.not_empty()),
                   And(o2.not_empty(),
                       o1.ip.dst == o2.ip.dst,
                       o1.tcp.dst == o2.tcp.dst)))
```

Connection memorization (Property 7). The formalization of Property 7 uses the same inductive approach as in the ToyLB example. As shown below, the specification is decomposed into a base case and two inductive cases. The base case states that when a packet from internal network is forwarded to external world by MazuNAT, the translation will be still effective within the time window THRESHOLD.

```
def test_memorize_init(self):
    p0, p1, s0 = sym_pkt(), sym_pkt(), sym_state()
    o, s1 = handle_packet(s0, 'from_intern', p0)
    ext_port = o['to_extern'].tcp.src
    t = s0['rw'].curr_time
    ddl = t + THRESHOLD
    verify(Implies(is_tcp(p0),
                   steer_to(c, s1, p0, ext_port, ddl)))
```

Then, the two inductive cases show that processing a packet from other flows or any time event before the end of the time window do not effect existing translation mappings.

```
def test_memorize_step_pkt(self):
    p0, p1, s0 = sym_pkt(), sym_pkt(), sym_state()
    t = sym_time()

    p_diff = sym_pkt()
    ext_port = sym_port()
    _, s1 = handle_packet(s0, 'from_intern', p_diff)
    verify(Implies(And(steer_to(c, s0, p0, ext_port, t),
                       from_same_flow(p0, p1)),
                   steer_to(c, s1, p0, ext_port, t)))

def test_memorize_step_time(self):
    ext_port = fresh_bv('port', 16)
    p0, p1, s0 = sym_pkt(), sym_pkt(), sym_state()
    t0, t1 = sym_time(), sym_time()
    _, s1 = handle_time(s0, 'rw', t1)
    verify(Implies(And(steer_to(c, s0, p0, ext_port, t0),
                       z3.ULT(t1, t0),
                       from_same_flow(p0, p1)),
                   steer_to(c, s1, p1, ext_port, t0)))
```

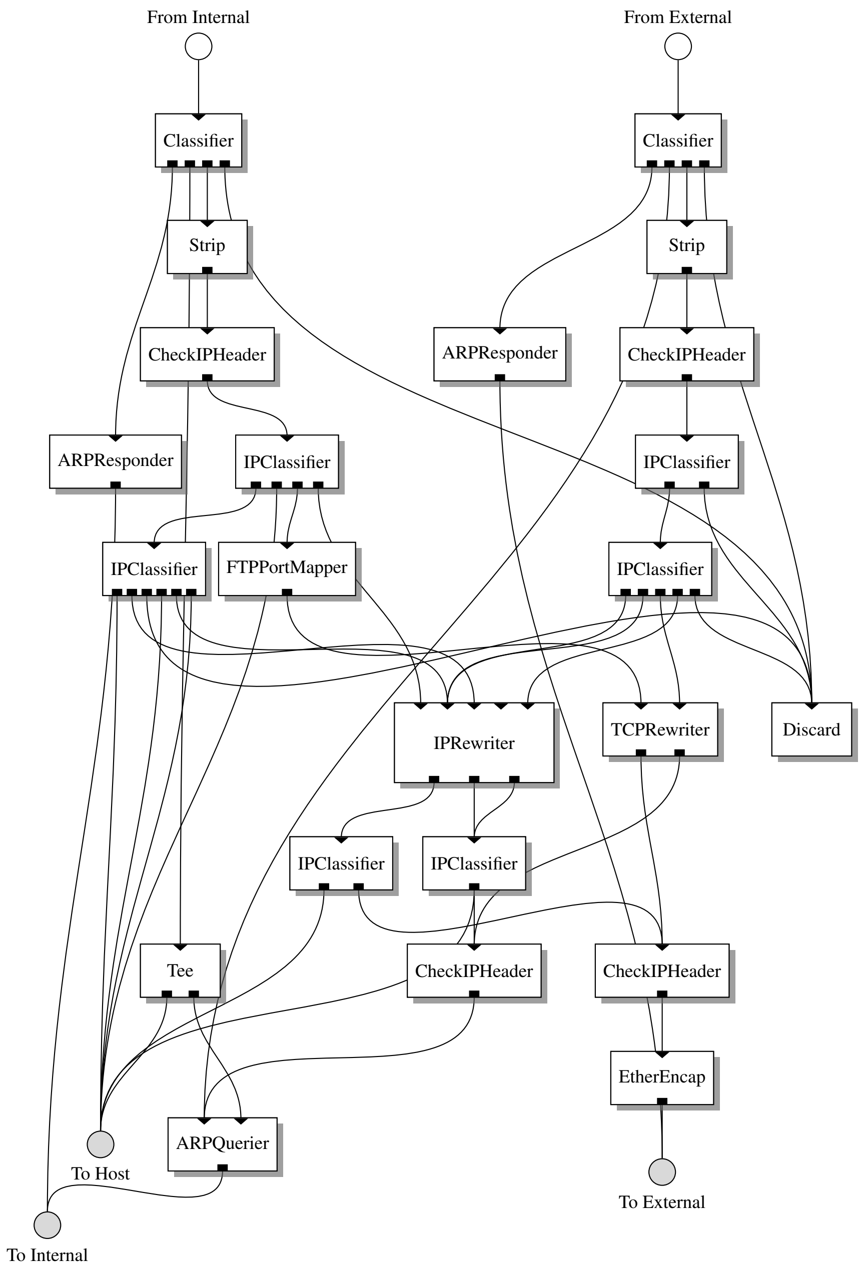


Figure 6: The directed graph of elements in MazuNAT.

APKeep: Realtime Network Verification for Real Networks

Peng Zhang*, Xu Liu*, Hongkun Yang†, Ning Kang*, Zhengchang Gu*, and Hao Li*
*Xi'an Jiaotong University, †Google

Abstract

Realtime network verification ensures the correctness of network by incrementally checking data plane updates in real time (e.g., $<$ 1ms per rule update). Even state-of-the-art methods can already achieve sub-millisecond verification time, such speed is achieved mostly for pure IP forwarding devices, and is unrealistic for real-world networks, due to two reasons. (1) Their network models cannot express the forwarding behavior of real devices, which have various functions including IP forwarding, ACL, NAT, policy-based routing, etc. (2) Their update algorithms do not scale in space and/or time: multi-field rules (e.g., ACL rules) can make these tools run out of memory and/or incur long verification time. To scale realtime verification to real networks, we propose APKeep based on a new modular network model that is expressive for real devices, and propose new algorithms that can achieve low memory cost and fast update speed at the same time. Our experiments show that for real-world update traces consisting of IP forwarding rules and ACL rules, existing methods either run out of memory or incur a prohibitively long verification time, while APKeep still achieves a sub-millisecond verification time. We also show that APKeep can verify an update of NAT rule mostly in less than 1 millisecond.

1 Introduction

Computer networks are prone to faults due to protocol misconfigurations, software bugs, and hardware failures [7,17,26,35]. Manually troubleshooting the faults often costs a network downtime up to several hours [7]. How to prevent network faults by ensuring network correctness becomes a fundamental problem posed to network operators and researchers.

Network verification seeks to automatically check network correctness at both control plane [9, 12, 13, 16, 18, 19, 30] and data plane [10, 15, 20, 22–24, 27, 36–40]. Compared to control plane verification which focuses on detecting protocol misconfigurations, data plane verification directly checks the data plane, which is closer to the actual forwarding behaviors of packets, and thus can catch a broader range of faults due to switch software bugs and hardware failures.

More recently, *realtime data plane verification* allows operators to check the correctness of data plane as it updates in realtime [20, 22, 24, 37–39]. To achieve this, realtime data plane verifiers often partition packets into *equivalence classes (ECs)*, and maintain a model of forwarding behavior for these ECs. When the data plane updates, they *incrementally* update the model, and check the updated model against correctness properties.

State-of-the-art realtime data plane verifiers have already achieved sub-millisecond verification time [20, 24]. However, such speed is mostly achieved for pure forwarding devices. For real devices consisting of various functions other than forwarding, these verifiers exhibit two fundamental limitations.

Network model is *not expressive* for real devices. Apart from IP forwarding, real devices have many other functions including access control list (ACL), network address translation (NAT), etc., which are composed in specific orders to implement various processing logic. For example, inside a typical router, multiple ACLs can be chained and applied at multiple ports to filter inbound and/or outbound packets [1]. Some routers may perform NAT on packets matching an ACL. Tools like VeriFlow [24] and Delta-net [20] assume simple models which only express forwarding functions. Models of NetPlumber [22] and AP Verifier [38, 39] can express more functions, but are hard to extend. For example, most vendors provide variants of policy-based routing [8], and adding such a feature requires heavy modification of their models. Even for the same set of functions, different devices may also have different pipelines, and writing a model for each of them is clearly not scalable.

Verification algorithms are *not scalable* for real devices. Range EC-based methods like VeriFlow and Delta-net represent each EC as a range of packet headers, thereby achieving a fast verification speed for IP forwarding rules. For example, Delta-net can check an update of IP forwarding rule in tens of microseconds on average. However, when there are multi-field rules, e.g., ACL rules, range EC-based methods may suffer from the problem of *EC explosion*, where the number of ECs grows exponentially with the number of multi-field rules. We find that for a real-world dataset consisting of only 686 ACL rules, an open-source version of VeriFlow and our multi-field extension of Delta-net can create up to 15 million ECs, causing either prohibitively long verification times or memory overflows.

AP Verifier computes the minimum number of ECs with respect to the network behavior. The downside, however, is that the update of ECs is more difficult, and can cost up to 10 milliseconds [37].

To overcome the above limitations and bring realtime network verification closer to the real world, this paper presents APKeep, a new realtime data plane verifier.

APKeep builds on a new network model that is modular and expressive. It models networks in a granularity of logical functions instead of physical devices. Each function, e.g., forwarding, filtering, rewriting, is modeled as a logically-independent *element*, which holds a set of logical *ports* corresponding to different actions on packets. APKeep views packets forwarded to the same port (i.e., undergoing the same actions) at each element as an EC, and encodes each EC with a logical *predicate*. The modularity of our model makes it easy to support common functions and vendor-specific compositions of functions in real devices. In addition, it also reduces the update scope and makes the update more efficient.

APKeep uses novel algorithms to compute and maintain the minimum number of ECs in realtime. A key reason for EC explosion of existing methods is that they create ECs based on the match fields of rules, resulting in a lot of unnecessary ECs with the same forwarding behavior. In addition, they cannot compress these ECs after creation. APKeep significantly reduces the number of ECs based on two principles. *(1) Creating ECs only when necessary.* APKeep creates ECs only when it needs more ECs to express new forwarding behaviors. *(2) Merging ECs when possible.* APKeep tracks the forwarding behavior of each EC, and merges multiple ECs with the same forwarding behavior. *We proved that by applying the above principles, APKeep always maintains the minimum number of ECs during update.*

In summary, our contribution is three-fold:

- We introduce a new network model that is modular and expressive for modeling real network devices.
- We design APKeep, which uses novel algorithms to fast update the network model for realtime verification.
- We show APKeep achieves a sub-millisecond verification time for update traces consisting of IP forwarding rules, ACL rules, and NAT rules.

Roadmap. We present the design overview (§ 2) and details (§ 3) of APKeep, followed by a case study (§ 4). Then, we show the experiment results (§ 5). After discussing related work (§ 6) and potential issues (§ 7), we conclude (§ 8).

2 Design Overview

This section overviews the design of APKeep. We will first introduce the network model that APKeep builds on, and then show how APKeep can fast update the model.

2.1 The Modular Network Model

To achieve realtime network verification for real networks, the network model should satisfy three key requirements: (1) *expressive* for common functions in real devices, e.g., IP forwarding, ACL, NAT, policy-based routing, etc.; (2) *extensible* for different devices with different vendor-specific implementations of these functions; (3) *efficient* to update for achieving realtime verification.

We propose *Port-Predicate Map (PPM)*, a new network model that meets all the above requirements. To demonstrate how PPM works, we use the example network shown at the top-left corner of Figure 1. In this network, switch *C* has four functions or modules (two ACLs, one forwarding, and one NAT), each having its own rules. If packets arrive at *port*1, two ACLs *ACL1* and *ACL2* are applied in sequence; if they arrive at *port*2, only *ACL1* is applied. Then, the packets will be sent to an output port according to the forwarding rules. If the output port is *port*5, packets will go through an NAT.

As an alternative, we could model a device as a monolithic box. This approach has the following drawbacks. First, it will be difficult to extend the model for new functionalities. For example, a device from another vendor may have a different chaining of modules (e.g., NAT before IP forwarding), or a new function (e.g., overriding IP forwarding with user policies). Then, we need to compose another device model. In addition, it will also make the update inefficient. For example, suppose a rule is inserted into *ACL1*, then we need to update the ECs allowed by the two input ports.

Element. Instead of modeling a network as a set of devices, PPM models at a granularity of *element*, defined as a logically-independent function (e.g., IP forwarding, ACL, or NAT). Each element has its own set of *rules*, and holds a set of logical *ports*. Different from physical ports, i.e., interfaces, logical ports represent generic actions including "output to VLAN 10", "permit SSH traffic", "rewrite dstIP to 10.0.0.1". This allows elements to express a broad range of functions other than IP forwarding. In specific, an element holds one port for each distinct action of rules in the element, and a special port for the default action is reserved for packets not matching any rules. When a packet arrives at an element, it will be "forwarded to" to exactly one port of the element, i.e., taking the actions of that port. Currently, PPM supports three types of elements, and more types can be added in the future.

- A **forwarding element** has rules that match IP prefixes and whose actions are "output packets to a specific set of interfaces". A forwarding element holds one port for

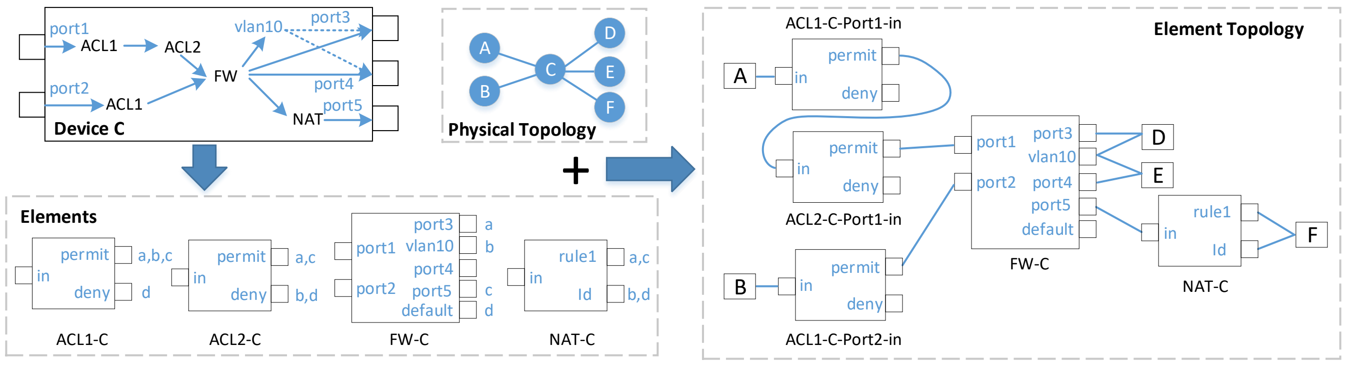


Figure 1: An example showing how APKeep divides devices into elements.

each distinct set of interfaces, and a *default* port for the default action, e.g., dropping packets.

- A **filtering element** has rules that match 5-tuples and whose actions are either "permit" or "deny". A filtering element holds exactly two ports: *permit* and *deny*.
- A **rewriting element** has rules which match 5-tuples and whose actions are "rewrite a specific header field to a specific value". A rewriting element holds one port for each distinct rewriting action, and an *id* port corresponding to no packet rewrite.

When a device has multiple functions, we break it into multiple elements. As shown in the left bottom of Figure 1, device C breaks into a forwarding element `FW-C`, two filtering elements `ACL1-C`, `ACL2-C`, and a rewriting element `NAT-C`.

Equivalence Class. Let $\mathcal{E}$ be the set of all elements in the network, and $\mathcal{H}$ be the set of all packet headers. For each header $h \in \mathcal{H}$ and element $e \in \mathcal{E}$, let $Port_e(h)$ be the port that h would be "forwarded to", assuming h has been received by e. Then, we have the following definition for equivalence class (EC).

Definition 1. *We say $\mathcal{C} = \{c_1, c_2, \dots, c_n\}$ is a set of equivalence classes (ECs) with respect to element set $\mathcal{E}$ and header set $\mathcal{H}$ if: (1) $c_i \wedge c_j = \emptyset, i \neq j$; (2) $\vee_{i=1}^{n} c_i = \mathcal{H}$; (3) $\forall h_1, h_2 \in \mathcal{H}$, $h_1 \neq h_2$: $\exists c \in \mathcal{C}$, $h_1, h_2 \in c \Rightarrow \forall e \in \mathcal{E}, Port_e(h_1) = Port_e(h_2)$ [1]. We say $\mathcal{C}$ is the minimum set of ECs if it is the smallest set satisfying the above conditions.*

APKeep encodes an EC with a logical *predicate*, i.e., Boolean formula. The reason to use predicate instead of range as in [20, 24] is that a predicate can encode an arbitrary set of packet headers, such that multiple range-based ECs having the same forwarding behavior can be represented as a single predicate. This allows APKeep to merge ECs with the same forwarding behavior, thereby avoiding explosion of ECs (§ 2.2).

Port-Predicate Map. For each predicate c and each element e, let $Port_e(c) = Port_e(h), \forall h \in c$. Suppose $p = Port_e(c)$, then we say port p *holds* predicate c. Define the *predicate set* of port p as: $Pred(p) = \{c \in \mathcal{C} | Port_e(c) = p, e \in \mathcal{E}\}$. We can see that $Pred$ is a map from port to predicates, which encodes the network forwarding behavior: given a packet h at element e, suppose it belongs to predicate c, then h will be forwarded by e to the port p satisfying $c \in Pred(p)$.

Element Topology. PPM uses the *element topology* to describe how elements are chained to process packets in the network. The right of Figure 1 shows the element topology of the example. First, each node represents an "application" of the corresponding element. For example, since `ACL1-C` is applied to *port*1 and *port*2, there are two nodes `ACL1-C-Port1-in` and `ACL1-C-Port2-in`. The forwarding element `FW-C` is applied once, and thus it corresponds to a single node. Creating a separate node for each application allows elements to be agnostic of input ports where packets are received. Second, each node has a set of ports, each holding a set of predicates, in the same way as its corresponding element. Thus, we only need to update a single element rather than all its nodes. For example, when a rule is inserted into $ACL1$, we only update the element `ACL1`, rather than its two nodes. Third, nodes are connected based on the physical topology, and how the elements are applied inside devices. For example, a port of A is connected to *port*1 of C in the network topology. Then, in the element topology, the port of A connects to the *in* port of `ACL1-C-Port1-in`, whose *permit* port connects to the *in* port of `ACL2-C-Port1-in`, and its *permit* port connects to the *port*1 of `FW-C`. The element topology will be used to construct forwarding graphs for verification (§ 3.3).

As shown above, PPM achieves modularity by breaking the composite functions inside a device into logically-independent elements. This brings the following benefits.

Expressiveness. Using the three types of elements as building blocks, PPM can express the forwarding, ACL, and NAT functions. Besides that, we will show how PPM can express the policy-based routing function offered by a major device vendor (§ 4).

Extensibility. Even most devices share roughly the same set of functions, the implementations and compositions of these functions are often vendor-specific. Writing a model for each

[1] Condition (3) says that for each h_1 and h_2 in $\mathcal{H}$ such that $h_1 \neq h_2$, we have: if there exists an c in $\mathcal{C}$ such that h_1 and h_2 both belong to c, then for each element e in $\mathcal{E}$, $Port_e(h_1) = Port_e(h_2)$

different device wastes time and effort. PPM models each device at the function level with elements, therefore it is relatively easy to model devices with vendor-specific compositions of functions by properly chaining the elements.

Reduced update scope. First, updates of multiple elements are decoupled, and when a rule is updated, we only need to update the element where the rule is updated, without affecting other elements. For example, multiple ACLs may be chained and applied to an interface. If a rule is inserted to one ACL, we only need to update the element of that ACL. Secondly, the application of elements is decoupled away from the elements themselves. For example, an ACL can be applied to multiple interfaces, and we only need to update the element of the ACL once, instead of updating all these interfaces. As another example, an operator may activate/deactivate an existing ACL on a port, or even migrate an ACL from a port to another [33]. In this case, we do not need to update the element of the ACL, as the forwarding behavior of the element is not affected.

2.2 The Update of Network Model

In the following, we show how APKeep updates the network model using a simple example in Figure 2. As shown in (a), the device has an ACL applied to its input port, followed by a forwarding module. For simplicity, we assume there are two match fields: *dstIP* represented with 2 bits x_1, x_2, and *dstPort* represented with 2 bits y_1, y_2. The forwarding module matches only *dstIP* with longest prefix match, and the ACL matches both *dstIP* and *dstPort* according to priorities (larger number means higher priority). We assume that by default, the ACL denies all packets and the forwarding module forwards all packets to *port*1. Initially, we have one EC *a*, which appears at the port *port*1 of element `FW`, and the port *deny* of element `ACL`. We will insert two ACL rules $R1$ and $R2$ shown in (b), and two forwarding rules $R3$ and $R4$ shown in (c).

First, we insert an ACL rule $R1$, whose match fields are $x_1x_2 = 0*, y_1y_2 = 00$, as shown at the top of (d). APKeep analyzes how $R1$ will affect the behaviors of element `ACL`. Specifically, APKeep finds $R1$ overrides the default deny rule in the red dashed rectangle. However, since $R1$ also has a deny action, packets in the rectangle will not change. Thus, APKeep does not update the EC *a*, which still appears at *port*1 of `FW` and *deny* of `ACL`, as shown at the bottom of (d). In contrast, if we create range-based ECs based on match fields, we will split EC *a* into three ECs, each of which is a rectangle in the header space.

Suppose another ACL rule $R2$ is inserted. Since $R2$ has a lower priority than $R1$, APKeep finds $R2$ can match only the shaded area, where it overrides default deny rule. As a result, packets matching the shaded area will change their behavior from deny to permit. To reflect that change, APKeep decides to transfer those packets from port *deny* to port *permit*. Since the packets are a portion of EC *a*, it splits *a* into two ECs, i.e., *b* for the shaded area, and another one by subtracting *b* from *a*. Then, APKeep transfers *b* to port *permit*, as shown at the bottom of (e). The reason that the EC *b* can be a non-rectangle area is that ECs are encoded with predicates. Specifically, the match fields of $R1$ and $R2$ can be represented as predicates $\bar{x}_1\bar{y}_1\bar{y}_2$ and $\bar{y}_1$, respectively. Then, *b* can be calculated by logical operations as $b = \bar{y}_1 \wedge \neg(\bar{x}_1\bar{y}_1\bar{y}_2) = (\bar{y}_1x_1) \vee (\bar{y}_1y_2)$.

Suppose a forwarding rule $R3$ is inserted. $R3$ overrides the default rule in the red dashed rectangle, which changes its port from *port*1 to *port*2. APKeep creates two new EC *c* and *d* by splitting *a* and *b*, respectively, and transfers them to *port*2, as shown in (f). The insertion of another forwarding rule $R4$ is similar and shown in (g). At this time, APKeep finds two ECs *d* and *f* appear at the same port at both elements, meaning that they have the same forwarding behavior – permitted by the ACL and forwarded to *port*2. Thus, APKeep merges *d* and *f* into a single EC. Similarly, APKeep merges *c* and *e* into a single EC, as shown in (h). The merging of ECs translates into logical disjunction of predicates. For example, *d* and *f* are merged into an EC represented by $(\bar{x}_1\bar{x}_2\bar{y}_1y_2) \vee (x_1\bar{x}_2\bar{y}_1)$.

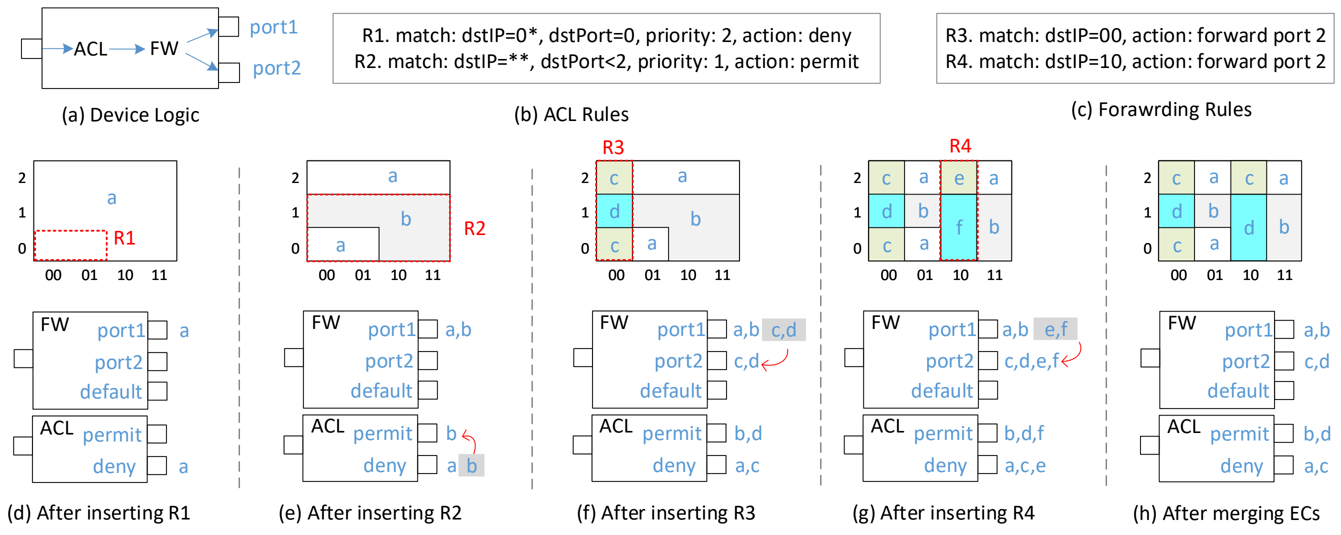
Finally, after inserting $R1$ through $R4$, APKeep creates 4 ECs. In contrast, if we create range-based ECs based on match fields, we will need 10 ECs, one for each rectangle of (h). The above is just an over-simplified example with only two fields which have 3-4 values. In real scenarios, the reduction rate can be as high as 99.99% (see Table 3). Actually, we prove that APKeep always maintains the minimum number of ECs during update (see Theorem 1).

The reason that APKeep can update such a small number of ECs is two-fold: *(1) Creating new ECs only when necessary.* APKeep creates a new EC only when part of an existing EC changes its forwarding behavior and the EC needs to be split into two ECs. In contrast, creating new ECs whenever the match fields of the new rule split some existing ECs will result in many redundant ECs. *(2) Merging ECs whenever possible.* APKeep tracks the forwarding behaviors of ECs, and merges multiple ECs if they have the same forwarding behavior. In contrast, range-based EC presentation mostly does not allow ECs to be merged.

The update of ECs in APKeep is much faster compared to AP Verifier due to the following reason. APKeep can quickly identify the changes of forwarding behaviors, and incrementally update predicates instead of re-computing them (§ 3.2). In contrast, AP Verifier maintains a port predicate for each device port, and computes atomic predicates (minimum number of ECs) based on all port predicates. When a rule is updated, it needs to first update the port predicates, and if new port predicates are created, it re-computes the atomic predicates based on the updated port predicates. We observe an up to 200× speedup in our experiments.

3 Design Details

This section presents the design of APKeep. Figure 3 shows the architecture of APKeep, which consists of three layers:



ACL
FW
port1
port2
(a) Device Logic
R1. match: dstIP=0*, dstPort=0, priority: 2, action: deny
R2. match: dstIP=**, dstPort<2, priority: 1, action: permit
(b) ACL Rules
R3. match: dstIP=00, action: forward port 2
R4. match: dstIP=10, action: forward port 2
(c) Forawrding Rules
R1
R2
R3
R4
default
permit
deny
(d) After inserting R1
(e) After inserting R2
(f) After inserting R3
(g) After inserting R4
(h) After merging ECs

Figure 2: An example of incremental update for rule insertion.

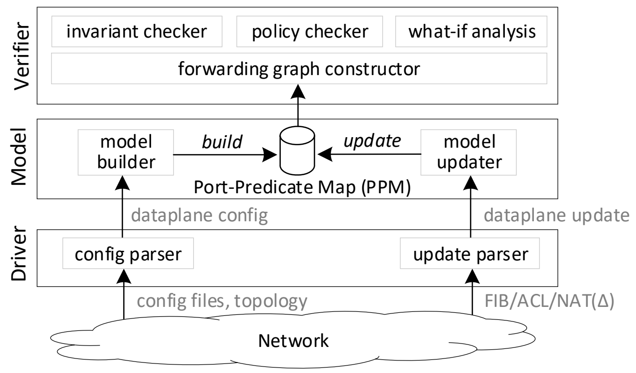


Figure 3: The architecture of APKeep.

The driver layer serves as the interface between network data plane and the model layer. In the bootstrap stage, the *config parser* reads in the network topology and configuration files, and generates the vendor-neutral *data plane config*, describing the configuration of interfaces, VLANs, ACLs, etc. for each device. The *update parser* fetches the FIB/ACL/NAT (changes) from each device and generates *data plane updates*, including insertion/deletion/modification of rules.

The model layer is the core of APKeep system. The *model builder* constructs PPM model by creating all the elements based on the data plane config. The *model updater* continuously updates the PPM model by processing each data plane update in sequence.

The verifier layer hosts verification applications on top of the model layer. The *forwarding graph constructor* generates forwarding graphs based on the PPM model, and on top of the graphs, various applications can be deployed to check network invariants, operator policies, or conduct what-if analysis.

In the following, we show how APKeep builds and updates the PPM model, and performs verification. Then, we show how APKeep supports packet rewrites, and present some optimization techniques.

3.1 Building PPM

For each device, APKeep constructs a device model based on its configuration of interfaces, ACLs, NAT, etc., and decomposes the device model into a set of elements. Currently, APKeep offers three types of elements, i.e., forwarding element, filtering element, and rewriting element. Initially without any rules, a forwarding element has a $default$ port; a filtering element has a *permit* port and a *deny* port; a rewriting element has an *id* port. For forwarding and rewriting elements, more ports can be created on-the-fly during rule insertions.

After creating elements, APKeep constructs the *element topology* by augmenting the physical topology with intra-device element connections, based on how elements are composed inside the device. For example, if an ACL $ACL1$ is declared to filter inbound traffic at port $port1$, then there is a connection from the *permit* port of $ACL1$ to the $port1$ port of the forwarding element.

Initially there is only one `True` predicate, standing for the set of all possible packets. For each element, the `True` predicate is held by its $default$, *deny*, or *id* port, depending on the element type. APKeep initializes the predicate set $Pred(p)$ (§ 2.1) to {`True`} if p is $default$, *deny*, or *id* port, and to empty set otherwise.

3.2 Updating PPM

For each rule update, APKeep updates the PPM using three steps: (1) encoding the match fields of the rule, (2) identifying the changes of forwarding behavior, and (3) updating the predicates and the map from port to predicates. The following only shows the case for rule insertion, and rule deletion differs only slightly in Step (2). Rule modification can be seen as a pair of rule deletion and insertion.

Let r be the rule to be inserted, specified as a 3-tuple $(priority, match, action)$, and let e be the element where r

Algorithm 1: `IdentifyChangesInsert`(r, $\mathcal{R}$)

```
Input: r: the newly inserted rule; R: the list of existing rules,
       sorted by decreasing priorities.
Output: C: the set of changes due to the insertion of rule r.
1  $C \leftarrow \{\}$;
2  $r.hit \leftarrow r.match$;
3  foreach $r' \in \mathcal{R}$ do
4      if $r'.prio > r.prio$ and $r'.hit \wedge r.hit \neq \emptyset$ then
5          $r.hit \leftarrow r.hit \wedge \neg r'.hit$;
6      if $r'.prio < r.prio$ and $r'.hit \wedge r.hit \neq \emptyset$ then
7          if $r'.port \neq r.port$ then
8              $C \leftarrow C \vee \{(r.hit \wedge r'.hit, r'.port, r.port)\}$;
9          $r'.hit \leftarrow r'.hit \wedge \neg r.hit$;
10 Insert $r$ into $\mathcal{R}$;
11 return $C$;
```

is inserted.

Step 1. Encoding match fields. Assume each packet header has h bits, each of which can be represented as a Boolean variable. Then, the match field of a rule corresponds to a set of packet headers, and can be represented as Boolean formula of h variables. For example, an IP match field of 128.0.0.∗ can be represented as $x_1 \wedge \bar{x}_2 \wedge \cdots \wedge \bar{x}_{24}$. We adopt the methods of [37] to encode the Boolean formulas of match fields based on Binary Decision Diagram (BDD [11]). BDD is a data structure that can canonically represent Boolean formulas, and it allows efficient logical operations including conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). By encoding the match fields with BDDs, we can efficiently compute and update predicates leveraging these logical operations. We use *r.match* to denote the match fields of r, encoded with BDD.

Step 2. Identifying changes. This step identifies the changes of forwarding behavior at element e, by analyzing how the insertion of r affects existing rules of e. Here, a behavior change takes the form of $(\delta, from, to)$, meaning packets satisfying predicate δ, which are originally forwarded to port $from$, will now be forwarded to port to. Note that this step is locally performed at e.

Before introducing the algorithm, we define the *hit* and *port* fields for each rule. First, note that multiple rules may have overlapping match fields, and packets will take the action of the rule with the highest priority. Thus, some headers in *r.match* may not "hit" rule r due to the presence of some higher-priority rules. To represent the headers that actually "hit" a rule, we define the *hit field* for each rule r as:

$$r.hit \triangleq \neg(\vee_{r'.prio > r.prio} r'.match) \wedge r.match \qquad (1)$$

If $h \in r.hit$, we know that h will take the action of r. Initially when there is only one default rule, the hit field of the default rule is equal to its match field, i.e., `True`. Second, recall that each element has a port corresponding to each distinct action of rules in the element. We use $r.port$ to denote the port corresponding to the action field of r. As an example, if r is a forwarding rule whose action field is "output to interface eth0/0", then $r.port = eth0/0$.

Algorithm 1 summarizes the procedure to identify the set of all behavior changes when a rule is inserted. It calculates the hit field *r.hit* by subtracting the match fields of higher-priority rules from *r.match* (Line 3-5), and identifies all behavior changes by analyzing how *r.hit* "overrides" lower-priority rules with different ports (Line 6-9). The algorithm for rule deletion differs only slightly and is not given here.

Step 3. Updating predicates. In this stage, APKeep takes the set of behavior changes caused by the inserted rule, denoted by C, and computes the set of transferred predicates, denoted by D. The process is summarized in Algorithm 2. In order to track which ports hold a given predicate, the algorithm maintains a map *Port* from each predicate c to the set of ports holding c, defined as $Port(c) = \{Port_e(c) | e \in \mathcal{E}\}$. We term $Port(c)$ as the *port set* of predicate c.

Initially, the set of transferred predicates D is set to empty (Line 1). For each change $(\delta, from, to)$, we iterate over each predicate p in the predicate set of $from$, and check whether p overlaps with δ (Lines 2-4). If so, we further perform the following three steps (Lines 5-10).

(1) Splitting predicates. In this step, we check whether p belongs to δ (Line 5). If not so, we need to split p into two new predicates $p \wedge \delta$ and $p \wedge \neg\delta$. by invoking the `Split` function (Line 6). As shown in Lines 11-17, the function `Split`$(p, p1, p2)$ first updates the predicate set of each *port* in $Port(p)$, by replacing p with $p1$ and $p2$ (Lines 12-13). Then, it initializes the port set of $p1$ and $p2$ with that of p (Lines 14-15). Finally, it updates the set of transferred predicates if needed (Lines 16-17).

(2) Transferring predicates. This step transfers the predicate $p \wedge \delta$ from port $from$ to port to by invoking the `Transfer` function (Line 7), as shown in Lines 18-22.

(3) Merging predicates. This step checks whether each predicate p' held by port to has the same port set with p (Line 8). If so, p' and p have the same forwarding behavior, and we merge them into a new predicate $p \vee p'$, by invoking the `Merge` function (Line 9), as shown in Lines 23-28.

After the above three steps, we update δ by subtracting p from it, and proceed to the next predicate of port $from$ (Line 10).

Theorem 1. *APKeep maintains the minimum set of equivalence classes after each rule update.*

The proof is given in Appendix A.

3.3 Verification

Checking Invariants. APKeep can check network invariants including loop-freedom and blackhole-freedom, which are defined as follows.

Algorithm 2: `Update`(C)

```
Input: $C$: the set of changes identified in the first stage.
Output: $D$: the set of transferred predicates.
1  $D \leftarrow \{\}$;
2  foreach $(\delta, from, to) \in C$ do
3      foreach $p \in Pred(from)$ do
4          if $p \wedge \delta \neq \emptyset$ then
5              if $p \wedge \delta \neq p$ then
6                  Split($p, p \wedge \delta, p \wedge \neg\delta$);
7              Transfer($p \wedge \delta, from, to$);
8              if $\exists p' \neq p, Port(p') = Port(p)$ then
9                  Merge($p, p', p \vee p'$);
10             $\delta \leftarrow \delta \wedge \neg p$;
11 Function Split($p, p1, p2$):
12     foreach $port \in Port(p)$ do
13         $Pred(port) \leftarrow Pred(port) \cup \{p1, p2\} \setminus \{p\}$;
14     $Port(p1) \leftarrow Port(p)$;
15     $Port(p2) \leftarrow Port(p)$;
16     if $p \in D$ then
17         $D \leftarrow D \cup \{p1, p2\} \setminus \{p\}$;
18 Function Transfer($p, from, to$):
19     $Pred(from) \leftarrow Pred(from) \setminus \{p\}$;
20     $Pred(to) \leftarrow Pred(to) \cup \{p\}$;
21     $Port(p) \leftarrow Port(p) \cup \{to\} \setminus \{from\}$;
22     $D \leftarrow D \cup \{p\}$;
23 Function Merge($p1, p2, p$):
24     foreach $port \in Port(p1)$ do
25         $Pred(port) \leftarrow Pred(port) \cup \{p\} \setminus \{p1, p2\}$;
26     $Port(p) \leftarrow Port(p1)$;
27     if $p1 \in D$ or $p2 \in D$ then
28         $D \leftarrow D \cup \{p\} \setminus \{p1, p2\}$;
29 return $D$;
```

- **Loop.** A packet traverses the same device for the second time, without being modified.
- **Blackhole.** A packet arrives at a device but does not match any forwarding rule.

Similar to Delta-net, APKeep checks invariants by constructing and traversing a *delta forwarding graph (DFG)*, a graph with each edge labeled with the ECs allowed on the edge. The difference is that APKeep updates the PPM model rather than DFG, and only constructs DFG based on the PPM model when checking invariants. Specifically, given a set of transferred predicates, APKeep constructs the DFG by adding the transferred predicates and the corresponding edges on the element topology. Then, APKeep traverses the DFG with a set of predicates P, which is initialized to the transferred predicates. When an edge is visited, P is intersected with the set of predicates on that edge. The traversal terminates when P becomes empty, reaching an edge with no next hop (blackhole detected), or the same node is visited twice (loop detected). The construction and traversal algorithms of DFG are given in Appendix B.

Checking policies. Operators may need to check user-defined policies such as hosts in a specific prefix can or cannot access a web server, traffic from subnet1 to subnet2 should pass the firewall, etc. We show how APKeep can support this task. Here, we define a policy as a pair of *match condition* and *path constraint*, where the match condition can be specified by header fields (e.g., 5-tuple), and a path constraint can be specified by a regular expression. Given a policy, APKeep can convert its match condition into a *policy predicate*, i.e., a BDD denoted as q, and its path constraint into an automata denoted as A. APKeep can check whether the policy is satisfied after an update as follows.

Let D be the transferred predicates after an update. APKeep computes a new set of predicates $D_q \leftarrow \{\delta \in D | \delta \wedge q \neq false\}$, and constructs the DFG G_q based on D_q. Then, APKeep traverses G_q while updating an instance of automata A for each $p_i \in D_q$, denoted as A_i. Specifically, APKeep updates the automata A_i if the predicate p_i visits a new node in DFG. The policy is satisfied if after traversal, all the automata enter the absorbing states; otherwise, the policy is violated. Note here multiple policies can be checked in parallel, and for each policy, the updating of each automata can also be parallelized.

What-if analysis. Operators can use APKeep to conduct "what-if analysis", e.g., will the invariants break if a specific link fails? APKeep answers such a query by retrieving all the predicates traversing the link, constructing a DFG using these predicates, and traversing the DFG to check invariants. The time to answer such a query heavily depends on the total number of ECs. We will show APKeep achieves a much shorter running time than Delta-net (§ 5.4).

3.4 Supporting Packet Rewrites

APKeep supports packet rewrites with *rewriting elements*. A rewriting element consists of a list $\mathcal{T}$ of rewrite rules, where each $T \in \mathcal{T}$ matches on 5-tuples, and rewrites the header to a specific value. APKeep creates a port for each rule in the rewriting element.

Based on the match fields of rewrite rules, we can compute predicates, and assign them to each port of the rewriting element, just as the forwarding and filtering element. The different part is: (1) how to encode packet rewrites using logical operations; (2) how to update predicates in the presence of rewrites.

Encoding packet rewrites. we adopt the methods in [39] to encode packet rewrites with logical operations as follows. First, it uses the existential quantification on predicate. Let p be a predicate, and x be one of the Boolean variables that p is defined on. The existential quantification of x is defined as:

$$\exists x.p = p|_{x=true} \wedge p|_{x=false} \tag{2}$$

, where $p|_{x=true}$ sets the value of variable x in p as `true`. Suppose the header has two bits x_1, x_2, then an NAT rule T that rewrites it to $x_1 = 1, x_2 = 0$ can be encoded as a logical function:

$$T(p) = (\exists x_1 \exists x_2 . p) \wedge (x_1 \wedge \bar{x}_2) \tag{3}$$

The existential quantification operation is supported by BDD.

Updating predicates in the presence of rewrites. Recall that for verification, we need to traverse a DFG which is constructed based on PPM. When there are only forwarding and filtering elements, we only need to perform intersections on predicate sets during traversal. However, when there are rewriting elements, predicates need to be transformed, and we need to ensure two conditions:

(1) Each predicate should be unambiguously transformed, i.e., the transformation should be defined for each predicate in PPM. For example, suppose p is split into $p1$ and $p2$, we should know how to transform each of them; otherwise, when traversing with only $p1$ or $p2$ in the predicate set, the rewriting element does not know how to transform it.

(2) The result of transformation should be represented by a set of predicates in PPM such that the traversal can proceed. For example, suppose a predicate p is held by the port of rewrite rule T, and $T(p) = p'$. If p' cannot be represented by a set of predicates in PPM, the traversal cannot continue since p' is not "recognized" by other elements.

In order to satisfy these two conditions, we apply the following two operations: (1) when a predicate p of a rewriting port is split into $p1$ and $p2$, we compute $p1' = T(p1)$ and $p2' = T(p2)$, and apply operation (2). (2) if the transformation result p cannot be represented as a set of predicates, we create new predicates to represent p. Note that this may split a predicate of some rewriting port and trigger operation (1).

Algorithm 3 summarizes how APKeep handles rule updates for rewriting elements. First, it updates the predicates with Algorithm 2 (Line 1). The difference lies in that the algorithm also maintains a *rewrite table*, where for each entry (k, v), k is a predicate before rewrite, and v is a set of predicates after rewrite. After transferring one predicate p to the port of another rule r', we need to apply the rewrite rule r' on p, and ensure the values in the rewrite table are still predicates (Lines 2-12).

Algorithm 3: `UpdateRewrite`$(C, \mathcal{RT})$

```
Input: $C$: the set of changes identified in the first stage; $\mathcal{RT}$:
       the rewrite table.
Output: $D$: the set of transferred predicates.
1  $D \leftarrow$ UpdateRW($C$);
2  while $true$ do
3      $updated \leftarrow false$;
4      foreach $(k,v) \in \mathcal{RT}$ do
5          foreach $p \in v$ do
6              if $p \notin \mathcal{P}$ then
7                  foreach $p' \in \mathcal{P}$ do
8                      if $p' \wedge p \neq false$ and $p' \wedge \neg p \neq false$
                       then
9                          SplitRW ($p', p' \wedge p, p' \wedge \neg p$);
10                 $updated \leftarrow true$;
11     if $updated = false$ then
12         break;
13 Function SplitRW($p, p1, p2$):
14     Split($p, p1, p2$);
15     foreach $(k,v) \in \mathcal{RT}$ do
16         if $p \in v$ then
17             $v \leftarrow v \cup \{p1, p2\} \setminus \{p\}$;
18     $\mathcal{RT}.remove(p)$;
19     $\mathcal{RT}.add(p1, \{T(p1)\})$;
20     $\mathcal{RT}.add(p2, \{T(p2)\})$;
21 return $D$;
```

3.5 Optimization

Delayed predicate merging. In Algorithm 2, APKeep merges two predicates instantly if they have the same port set. However, for some datasets, we find that some predicates are repeatedly merged and split, resulting in a waste of time. Thus, we adopt a delayed predicate merging: when a predicate can be merged, we record it, and when the total number of predicates exceeds a threshold (500 by default), we merge all the recorded predicates. To fast determine whether a predicate can be merged, we maintain a hash table where the key is an ordered list of ports, and the value is a set of predicates that appear at all these ports.

Separate update for different types of elements. Updating both forwarding rules and ACL rules may result in a large number of predicates. For example, suppose there are n ECs generated by forwarding rules, and an ACL rule matching a destination port range will create n new ECs. AP Verifier [37] proposed to compute the atomic predicates for forwarding and ACL rules, separately. We adopt this approach and update two sets of predicates, one for forwarding elements, and one for ACL elements. When traversing the forwarding graph, we need to carry two sets of predicates, and set intersection only happen between the same set of predicates. Different from [37], our algorithm avoids false positives when verifying invariants. For example, when a node is visited twice, we evaluate whether there exist two predicates, one from each set, that have non-empty conjunction. If so, the loop exists; otherwise, the loop is a false positive.

4 Case Study

We study the expressiveness of our PPM model by showing how to model a vendor-specific function with the three built-in

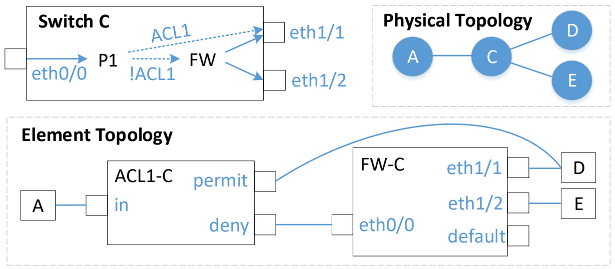


Figure 4: Modeling traffic policy in APKeep.

element types.

Policy-Based Routing (PBR) is a function commonly available in many routers and switches. It allows operators to override the IP forwarding rules such that packets are forwarded based on criteria other than destination IP address. Different vendors may implement their own version of PBR, and here we study one such implementation offered by a large device vendor.

The vendor offers a function named *traffic policy*, defined as a set of classifier-behavior pairs. The following shows a traffic policy `p1` applied to inbound traffic of interface `eth0/0` at switch *C*. `p1` is defined by a classifier `c1` and a behavior `b1`, meaning that packets satisfying `c1` will be forwarded according to `b1`. `c1` is defined using an ACL `ACL1`, and the behavior is redirecting traffic to interface `eth1/1`. The top-left and top-right of Figure 4 show the processing logic of switch *C* and the network topology, respectively.

```
interface eth0/0
traffic-policy p1 inbound
#
traffic policy p1 match-order config
classifier c1 behavior b1
#
traffic classifier c1 operator or precedence 5
if-match acl ACL1
#
traffic behavior b1
permit
redirect interface eth1/1
```

In our PPM model, the above traffic policy can be easily modeled by creating an ACL element `ACL1-C`, and properly chaining it into the forwarding graph, as shown in the bottom of Figure 4: (1) connecting its `in` port to the upstream port originally connected to `eth0/0`, (2) connecting its `permit` port to the downstream port originally connected to `eth1/1`, (3) connecting its `deny` port to the `eth0/0` port of `FW-C`.

The above is just the simplest form of traffic policy, and in a more general case, a policy can contain multiple classifier-behavior pairs, and each classifier can contain multiple ACLs. Then, we need to create multiple elements, one for each ACL, and cascade them together.

In addition to 5-tuples, a traffic policy also matches various information including VLAN ID, layer-3 packet length, time ranges, etc. Since the predicate-based EC representation has no restriction on the match fields, we can encode these match conditions by adding more fields. For example, we can add a 16-bit field to encode the packet length from 0 to 65535, and a 5-bit field to encode the hour-level time range. Note since PPM models the packet forwarding behaviors of symbolic packets, the fields to add do not have to be packet headers.

Apart from PBR, the traffic policy function also supports other behaviors including traffic statistics, flow mirroring, etc., which do not change the forwarding behaviors, and rate limiting, congestion avoidance, which selectively drop packets. As all previous data plane verifiers, PPM cannot model these features.

Table 1: Dataset statistics.

Network	Nodes	Links	Forwarding rules	ACL rules	Updates
Airtel1	68	260	6.89×10^4	0	1.42×10^7
Airtel2	68	260	9.84×10^4	0	5.05×10^8
4Switch	12	16	1.12×10^6	0	1.12×10^6
Internet2	9	56	1.26×10^5	0	2.52×10^5
Stanford*	16	74	3.84×10^3	0	7.68×10^3
Purdue*	1,646	3,094	3.52×10^6	0	7.04×10^6
Stanford	124	182	3.84×10^3	686	9.05×10^3
Purdue	2,159	3,607	3.52×10^6	2,707	7.05×10^6

5 Evaluation

5.1 Setup

Implementation. We implemented APKeep with around 5K lines of Java code. Currently, we have implemented config parsers for three different vendors, which translate vendor-specific configuration files into a unified representation in JSON format. We also implemented an update parser for one vendor, whose devices support fetching data plane state including FIBs and ACLs. For verification, we implemented an invariant checker that can detect loop and blackhole, and a what-if analyzer that can reason about the possible impact of link failures. For BDD operations, we use JDD, a BDD library for Java [34].

Dataset. Table 1 shows the datasets we use. The first six consist of updates of IPv4 forwarding rules, and the last two consist of updates of both IPv4 forwarding rules and ACL rules. The first three datasets are generated by Delta-net [20] using the ONOS SDN-IP application [6], and the Internet2 dataset is from [5]. The Stanford dataset [2] consists of both IPv4 forwarding rules and ACL rules, and the original Purdue dataset [32] consists of only ACL rules. We generate forwarding rules for the Purdue dataset, using shortest path routing. Finally, we remove the ACL rules from these two datasets, and obtain another two pure-IP datasets Stanford* and Purdue*. Since the last five datasets are snapshots of rules, we generate a sequence of updates from each of them as follows. First, we add all the ACL rules (if any), one rule per ACL each time, and then all the forwarding rules, one rule per device each time. After that we delete these rules in the reverse order as they are inserted.

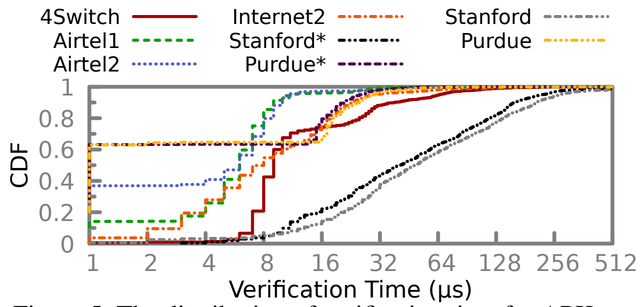


Figure 5: The distribution of verification time for APKeep.

Methods to compare. We compare APKeep with four data plane verification tools.

AP Verifier [37]. We use its open-source implementation in Java [4], and also the authors' implementation of incremental update algorithms [3]. We modify it to process our rule updates, and implement an incremental loop checker for it.

Delta-net [20]. We implement an extended version of Delta-net using C++, referred to as **Delta-net$^{\mathbf{MF}}$**. It handles single-field IP forwarding rules in the same way as Delta-net, and handles multi-field ACL rules using a multi-layered tree approach as in VeriFlow. Note, Delta-net^{MF} may not be the best approach for extending Delta-net so that it can apply to multiple fields.

VeriFlow [24]. We use its open-source implementation in C. Since VeriFlow only supports match fields expressed with prefixes, an ACL rule matching port ranges may be split into multiple ones which match prefixes.

NetPlumber [22]. We use its open-source implementation in C++ [2]. Since NetPlumber takes transfer function (TF) rule as input, we translate our rule updates into equivalent TF rule updates. Prior to update, we insert all the default rules created by NetPlumber since the insertion and deletion of them take a long time, as confirmed by the paper. We attach one source node to each device for NetPlumber to inject “flows” in the network model.

Apart from the above four methods, we also consider **APKeep^{-}**, standing for APKeep without merging predicates. For benchmark purpose, we let each method check loops after each update. All the experiments run on a Linux desktop with a 3.0GHz Intel Core i5 CPU and 32GB RAM.

5.2 Verification Time

Figure 5 shows the verification time of APKeep. We can see for all datasets, the verification time is less than 250μs for 90% of updates. Table 2 compares the average running time of APKeep with the other methods. For datasets with only IP forwarding rules, the running time of APKeep is comparable to Delta-net^{MF}, and much shorter than the other methods. For the 4Switch dataset, APKeep is 253×, 128×, and 937× faster than AP Verifier and VeriFlow, and NetPlumber, respectively. Note that NetPlumber is relatively slow since it models each rule as a node, and computes all the flows through these rules. Thus, its model is more fine-grained than APKeep, but incurs a relatively high cost. Surprisingly, APKeep^{-} is even faster than APKeep on some datasets. The reason is that these datasets have a rather small number of ECs (see Table 3), and therefore merging ECs incurs additional overhead without paying off. However, for the 4Switch dataset, APKeep^{-} is much slower as it has 271,793 ECs, while APKeep has only 557 ECs. For datasets with multiple match fields, all other methods including APKeep^{-} either incur a prohibitively long running time or run out of memory. For the Purdue dataset, only APKeep runs to completion, with an average running time of 13μs; all other methods either time out or run out of memory. This demonstrates existing methods can hardly meet the realtime requirement when the rules to update match multiple fields.

5.3 Number of Equivalence Classes

We observe that the number of ECs heavily impacts the running time of realtime data plane verifiers. To confirm this, we report the number of ECs maintained by APKeep, APKeep^{-}, and Delta-net^{MF} in Table 3.

We can see that when there is a single match field, APKeep^{-} computes slightly fewer ECs than Delta-net^{MF}. The reduction is due to the fact that predicates can encode arbitrary packet sets, rather than ranges. By merging predicates, APKeep computes much fewer ECs than APKeep^{-}. This indicates that using predicates alone cannot efficiently reduce the number of ECs.

The number of ECs computed by Delta-net^{MF} grows from 2283 to 15 million after only 686 ACL rules are inserted in the Stanford dataset, and reaches over 100 million after 2,707 ACL rules are inserted in the Purdue dataset. Note that Delta-net^{MF} actually does not run to completion for Stanford and Purdue datasets, and the numbers are counted by running only the functions related to the creation of ECs. In contrast, APKeep computes only 515 and 4,160 ECs for these two datasets, a 99.99% reduction compared with Delta-net^{MF}.

The above results show that range-based EC representation easily leads to an explosion of ECs when there are only a small number of rules with multiple match fields. On the other hand, by representing ECs with predicates, and updating the minimum number predicates, APKeep can dramatically reduce the total number of ECs, thereby achieving a fast verification speed with small memory footprint.

Figure 6 shows the number of ECs maintained by APKeep and Delta-net^{MF} during the updates. The Airtel1 dataset consists of rule insertions and deletions which are generated to react to link failures. Thus, the number of rules is small during update, and the total number of ECs is also quite small. The 4Switch dataset only has rule insertions; and the last three datasets insert all rules and remove them later. Thus, for Internet2 and Stanford, APKeep finally has only one and two predicates, respectively. For the Purdue dataset, since both

Table 2: Average verification time of different methods (D^{MF} is shorthand for Delta-net^{MF}). TO means timeout ($>$ 24h), and MO means memory overflow ($>$ 32GB).

Network	Average verification time (μs)						Percentage $< 250\mu$s					
	AP Verifier	VeriFlow	NetPlumber	D^{MF}	APKeep$^-$	**APKeep**	AP Verifier	VeriFlow	NetPlumber	D^{MF}	APKeep$^-$	**APKeep**
Airtel1	80	59	3,804	3	5	**7**	91.3%	99.9%	3.8%	99.9%	99.9%	**99.8%**
Airtel2	135	48	TO	4	4	**6**	77.4%	99.9%	TO	99.9%	99.9%	**99.9%**
4Switch	5,316	2,706	19,678	4	2,190	**21**	7.8%	8.2%	0.8%	99.9%	75.1%	**99.8%**
Internet2	1,660	144	2,123	3	9	**12**	24.2%	93.3%	9.9%	99.9%	99.5%	**99.7%**
Stanford*	1,953	468	8,700	9	98	**94**	13.3%	96.1%	23.6%	99.9%	93.6%	**96.4%**
Purdue*	777	648	MO	15	2	**9**	83.7%	66.5%	MO	99.9%	99.9%	**99.9%**
Stanford	2,072	4.8×10^6	9,532	MO	3.1×10^5	**127**	24.3%	0.4%	34.0%	MO	11.8%	**91.7%**
Purdue	TO	TO	MO	MO	MO	**13**	TO	TO	MO	MO	MO	**99.8%**

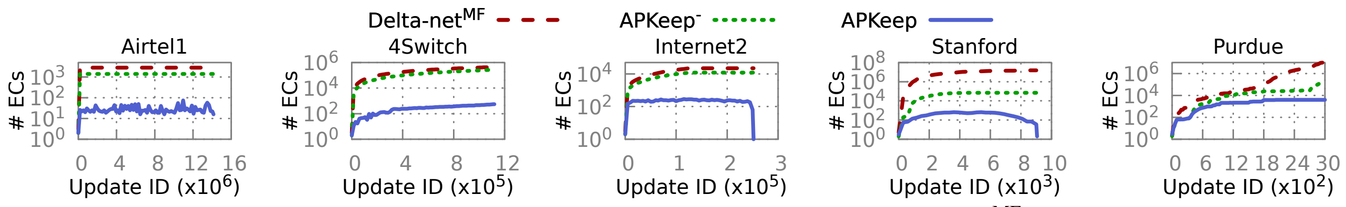


Figure 6: Number of equivalence classes maintained by APKeep and Delta-net^{MF} during updates.

Table 3: Number of ECs for APKeep and Delta-net^{MF}.

Network	Delta-net^{MF}	APKeep$^-$	APKeep	Reduction Rate
Airtel1	2,799	1,401	16	99.4%
Airtel2	2,799	1,401	64	97.8%
4Switch	443,443	271,793	557	99.9%
Internet2	22,212	14,819	216	99.0%
Stanford*	2,283	1,515	494	78.4%
Purdue*	1,176	939	267	77.4%
Stanford	15,100,968	842,734	515	99.99%
Purdue	>104,743,229	>168,891	4,160	99.99%

Table 4: Verification time for "what if" queries. The results for Delta-net were from paper [20], whose experiments ran on a 3.47GHz Intel Xeon CPU.

Network	# Rules	Average query time (ms)		+Loops (ms)	
		Delta-net	APKeep	Delta-net	APKeep
Airtel	38,100	0.04	0.02	2.3	0.13
4Switch	1,120,000	21.1	0.48	128.1	1.37

Delta-net^{MF} and APKeep$^-$ cannot run to completion, we only show the number for the first 3000 updates.

5.4 Answering "What if" Queries

We evaluate the running time for APKeep to answer "what if" queries. In particular, we consider the query "what is the fate of packets that use a link if the link fails?". To answer this query, Delta-net constructs a forwarding graph using those ECs on that link, and is reported to be 10× faster than VeriFlow. Thus, we only compare our results to those of Delta-net. In Table 4, Columns 3-4 show the average query time, and Columns 5-6 show the average query time if we additionally check loops. We can see that APKeep is 17× and 93× faster than Delta-net in overall query time for the Airtel and 4Switch datasets, respectively. The reason is that the number of ECs in APKeep is much smaller than that in Delta-net.

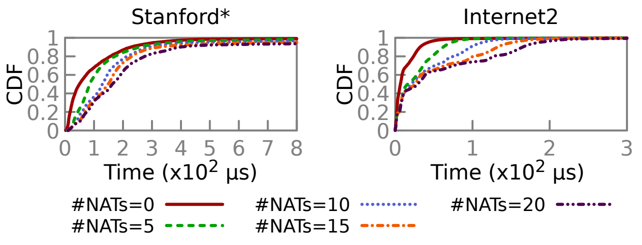


Figure 7: The cumulative distribution of verification time when different number of NATs are added.

5.5 Updating Rewrite Rules

We evaluate the time for APKeep to handle updates of rewrite rules. We use the Stanford* and Purdue* dataset, and add NAT rules into the network as follows. First, for each dataset, we find all the edge ports: an edge port holds a non-empty set of predicates, and is not connected to any other switches. Then, for each edge port, we randomly select a predicate associated with it, and compute an IP prefix that satisfies the predicate. Finally, for each IP prefix, we generate 25 NAT rules, each of which translates an IP address to another address belonging to a different IP prefix. We place the updates of NAT rules after the updates of forwarding rules.

Figure 7 shows the running time of APKeep for different numbers of NATs ranging from 0 to 20. Since each NAT has 25 rules, the number of NAT rules ranges from 0 to 1000. We can see that the running time of APKeep is mostly less than 1ms, and scales well with the number of NAT rules.

6 Related Work

Offline data plane verification was originally studied by Xie et al. [36], and later advanced by FlowChecker [10], Anteater [27], HSA [23], and NoD [25]. These tools take a snapshot of the data plane state, and check whether it satisfies network invariants like blackhole-freedom, loop-freedom, etc. AP Verifier [37–39] uses Binary Decision Diagraph (BDD) to compute a predicate for each port, and uses all port predicates to generate atomic predicates, which are the minimum set of ECs. APKeep differs in that it builds on a modular element-level model that is much more expressive than the monolithic model used by AP Verifier. In addition, APKeep incrementally updates the ECs instead of re-computing them from scratch, thereby achieving up to 200× speedup compared to the incremental version of AP Verifier (Table 2). To scale verification to large networks, Libra [40] uses MapReduce to parallelize verification. Plotkin et al. [29] propose to transform large networks into smaller ones for scalable verification, based on network surgery and symmetry. APKeep can leverage this technique to reduce network size, thereby scaling to larger networks. RCDC [15] decomposes data plane verification into the validation of local contracts. However, RCDC assumes structured datacenter networks so as to track the topology and address locality, while APKeep targets general networks. SymNet [31] and VMN [28] focus on verifying stateful data planes with middleboxes.

Realtime data plane verification incrementally checks the network data plane for each update in real time. NetPlumber [22] builds on the plumbing graph model, where each node is a rule and a flow is a set of packets traversing the same sequence of rules. Thus, the model has a finer grain than PPM, while the downside is that updating the model is relatively slow (Table 2). VeriFlow [24] achieves a smaller verification time ($<$ 1ms) by computing the equivalence classes (ECs) affected by an update, and checking the forwarding graph of each affected EC. Delta-net [20] further reduces the verification time by incrementally maintaining a single EC-labelled graph, rather than constructing multiple graphs for each update. VeriFlow and Delta-net can achieve sub-millisecond verification time for updates of single-field IP forwarding rules. Howerver, they may suffer from the problem of EC explosion when there are multi-field rules, and cannot handle updates of rewriting rules.

Representation of ECs. Bjørner et al. propose ddNF [14], a new data structure for representing ECs, and show it outperforms BDD on datasets consisting of forwarding rules. However, the set of ECs represented using ddNF may not be minimal. #PEC [21] introduces a new lattice-theoretic method which can construct the minimum number of ECs, faster than using BDD. #PEC may serve as a better foundation for multi-field extension of Delta-net, and it would also be interesting to study how to leverage #PEC to further speed up APKeep.

Control plane verification checks whether protocol configurations are correct [9, 12, 13, 16, 18, 19, 30]. They are orthogonal to APKeep, while tools like Batfish [18] may use APKeep to speed up the verification of generated data planes.

7 Discussion

Model modularity vs. number of ECs. Modeling the network at a fine granularity can make the update more efficient, while may also increase the number of ECs. The reason is that the model may have more different forwarding behaviors, which need to be represented with more ECs. For example, even two packets behave the same at a device level, their behaviors may differ in the intra-device processing, and thus should be represented with different ECs. Thus, there is a tradeoff between the model granularity and the number of ECs. It will be interesting to further navigate such tradeoff in the future.

Fetching data plane state. APKeep fetches the whole data plane state only once in the bootstrap stage, and only fetches data plane updates afterwards, whose cost can be much less compared to fetching the whole data plane. To ensure timeliness, APKeep needs to fetch the updates from devices at a sufficient frequency. We are aware some new devices have already provided APIs for fetching FIB updates, and we expect this feature will be supported by more devices in the future.

Ensuring update consistency. Since the data plane state is in continuous transition, a violation can be falsely triggered by a transient state. APKeep can be made robust to such inconsistency as follows. If an update fails the verification of an invariant, APKeep flags it as suspicious without raising an alarm. After a configured time window, APKeep checks the invariant again to confirm whether the update is a true violation. Detailed design is left as one of our future work.

Why microsecond-level verification. One major purpose of speeding up incremental verification is to ensure the network model, which verification is based on, can keep up with fast network updates. For example, in a large datacenter with 1k devices, a 1ms model update time only allows the verifier to keep up with an average network update rate of 1 update per device every second. Thus, further speeding up data plane verifiers can scale the verification to larger network size and higher data plane update rate.

8 Conclusion

This paper presented APKeep, a new realtime data plane verifier. APKeep builds atop PPM, a modular network model that is expressive for real devices, and incrementally maintains the minimum number of equivalence classes in realtime. We showed that for real updates consisting of both forwarding and ACL rules, all other methods either ran out of memory or incurred a prohibitively long verification time, while APKeep still achieved a sub-millisecond verification time.

Acknowledgments. We thank the anonymous NSDI reviewers and our shepherd Behnaz Arzani for their valuable feedback. We thank Zunying Qin for help on collecting device configurations. This work is supported by NSFC (No. 61772412 and 61702407) and the K. C. Wong Education Foundation.

References

[1] Configuring IP Access Lists - Cisco. https://www.cisco.com/c/en/us/support/docs/security/ios-firewall/23602-confaccesslists.html.

[2] Hassel, the header space library. https://bitbucket.org/peymank/hassel-public.

[3] Real-time Verification of Network Properties Using Atomic Predicates (technical report). http://www.cs.utexas.edu/users/lam/NRL/TechReports/Yang_Lam_TR-13-15.pdf.

[4] Scalable Network Verification Using Atomic Predicates. http://www.cs.utexas.edu/users/lam/NRL/Atomic_Predicates_Verifiers.html.

[5] The Internet2 Observatory. http://www.internet2.edu/research-solutions/research-support/observatory.

[6] The ONOS project. https://onosproject.org/.

[7] Troubleshooting the Network Survey. http://eastzone.github.io/atpg/docs/NetDebugSurvey.pdf.

[8] Understanding Policy Routing - Cisco. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/10116-36.html.

[9] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast and general network verification. In *USENIX NSDI*, 2020.

[10] E. Al-Shaer and S. Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *ACM workshop on Assurable and usable security configuration*, 2010.

[11] H. R. Andersen. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen*, 1997.

[12] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, 2017.

[13] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. Control plane compression. In *ACM SIGCOMM*, 2018.

[14] N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia, and G. Varghese. ddNF: An efficient data structure for header spaces. In *Haifa Verification Conference*, 2016.

[15] N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma. Validating datacenters at scale. In *ACM SIGCOMM*, 2019.

[16] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *USENIX OSDI*, 2016.

[17] N. Feamster and H. Balakrishnan. Detecting bgp configuration faults with static analysis. In *USENIX NSDI*, 2005.

[18] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *USENIX NSDI*, 2015.

[19] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*, 2016.

[20] A. Horn, A. Kheradmand, and M. R. Prasad. Delta-net: Real-time network verification using atoms. In *USENIX NSDI*, 2017.

[21] A. Horn, A. Kheradmand, and M. R. Prasad. A precise and expressive lattice-theoretical framework for efficient network verification. In *IEEE ICNP*, 2019.

[22] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *USENIX NSDI*, 2013.

[23] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.

[24] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.

[25] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *USENIX NSDI*, 2015.

[26] R. Mahajan, D. Wetherall, and T. Anderson. Understanding bgp misconfiguration. In *ACM SIGCOMM*, 2002.

[27] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *ACM SIGCOMM*, 2011.

[28] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying reachability in networks with mutable datapaths. In *USENIX NSDI*, 2017.

[29] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. In *ACM POPL*, 2016.

[30] S. Prabhu, K.-Y. Chou, A. Kheradmand, P. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*, 2020.

[31] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM*, 2016.

[32] Y.-W. E. Sung, S. G. Rao, G. G. Xie, and D. A. Maltz. Towards Systematic Design of Enterprise Networks. In *ACM CoNEXT*, 2008.

[33] B. Tian, X. Zhang, E. Zhai, H. H. Liu, Q. Ye, C. Wang, X. Wu, Z. Ji, Y. Sang, D. Y. Zhang, Ming, C. Tian, B. Y. Zhao, and H. Zheng. Safely and automatically updating in-network acl configurations with intent language. In *ACM SIGCOMM*, 2019.

[34] A. Vahidi. JDD, a pure Java BDD and Z-BDD library. `https://bitbucket.org/vahidi/jdd/`.

[35] A. Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, 2004.

[36] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *IEEE INFOCOM*, 2005.

[37] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE ICNP*, 2013.

[38] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2016.

[39] H. Yang and S. S. Lam. Scalable verification of networks with packet transformers using atomic predicates. *IEEE/ACM Transactions on Networking*, 25(5):2900–2915, 2017.

[40] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *USENIX NSDI*, 2014.

A Proof of Theorem 1

Proof. According to Definition 1, it is easy to see that a set of ECs is minimum if the other direction of condition (3) also holds, i.e., if two packets are forwarded to the same port at each element, then they must belong to the same EC. Therefore, define (3)' by replacing "$\Rightarrow$" with "$\Leftrightarrow$" in (3), and we need to prove that conditions (1)(2)(3)' hold for all predicates in the PPM model.

Clearly, these conditions hold initially when there are no updates: each element has a single default rule, and there is only one `True` predicate, which is assigned to the default/deny/id port depending on element type. We prove the theorem by induction: if the conditions hold before an update, then they still hold after the update.

Let e be the element whose rule is updated. For condition (1)(2), it is clear that `transfer` operations do not modify predicates, and thus have no effect on these conditions. Also, since `split` and `merge` only move part of one predicate into another one, they will not break these conditions, either. In the following, we show both directions of (3)' hold.

$\Rightarrow$: Suppose $h_1, h_2 \in c$ after update, we show they appear in the same set of ports. There are two cases. (1) h_1 and h_2 belong to the same predicate before update. Then, we know $Port_s(h_1) = Port_s(h_2), \forall s$, and the update only changes $Port_e(h_1)$ and $Port_e(h_2)$. Then, we only need to show $Port_e(h_1) = Port_e(h_2)$ after update. Clearly this holds if neither h_1 or h_2 changes its port at e. Suppose at least one of them changes its port at e, and let $Port_e(h_1) = Port_e(h_2) = p_a$ before update. Without loss of generality, suppose h_1 changes its port to $p_b \neq p_a$, we prove by contradiction that h_2 must have also changed its port to p_b. Assume h_2 either keeps its port unchanged or changes its port to $p_c \neq p_b$. Suppose $h_1 \in c_1$ and $h_2 \in c_2$ after transferring predicates, then c_1 appears at port p_b, and c_2 appears at port p_a or p_c. Since c_1 and c_2 appear in different ports at e, they cannot be merged, contradicting our assumption that $h_1, h_2 \in c$ after update. (2) h_1 and h_2 belong to different predicates before update. Let $h_1 \in c_1$ and $h_2 \in c_2$ before update, then c_1 and c_2 must have been merged into c. That is, c_1 and/or c_2 must have been transferred to the same port after update, which implies that h_1 and h_2 must have changed their ports to the same one at e.

$\Leftarrow$: Suppose $Port_s(h_1) = Port_s(h_2), \forall s$ after update, there are two cases. (1) $h_1, h_2 \in c'$ before update. Then, $Port_s(h_1) = Port_s(h_2), \forall s$ before update, implying that both h_1 and h_2 do not change their ports or change to the same port. If they do not change their ports, then they will still belong to the same predicate (either c' if c' is not split or $c' \wedge \neg\delta$ if c is split). If they change their ports, then a predicate including h_1, h_2 (either c' if c' is not split or $c' \wedge \delta$ if c' is split) will be transferred to a new port, and they will still belong to the same predicate, no matter whether the transferred predicate is merged or not. (2) $h_1 \in c_1, h_2 \in c_2$ before update, then we know $Port_e(h_1) \neq Port_e(h_2)$ before update, and thus at least one of them must have changed its port. Without loss of generality, suppose h_1 has changed its port such that $Port_s(h_1) = Port_s(h_2)$ after the change, and let c' be the transferred predicate satisfying $h_1 \in c'$. This will trigger our algorithm to merge c' and c_2. $\square$

Algorithm 4: `ConstructDeltaForwardingGraph`(D)

```
Input: D: the set of transferred predicates.
Output: G: the forwarding graph used for invariant checking.
$V \leftarrow \{\}, E \leftarrow \{\}, A \leftarrow \perp$;
foreach $\delta \in D$ do
    foreach $port \in Port(\delta)$ do
        $s1 \leftarrow$ the node holding $port$;
        if $s1 \notin V$ then
            $V \leftarrow V \cup \{s1\}$;
        $s2 \leftarrow$ the node connected to $port$;
        if $s2 \notin V$ then
            $V \leftarrow V \cup \{s2\}$;
        if $(s1, s2) \notin E$ then
            $A(s1, s2) \leftarrow \{\}$;
            $E \leftarrow E \cup \{(s1, s2)\}$;
        $A(s1, s2) \leftarrow A(s1, s2) \cup \{\delta\}$;
return $G(V, E, A)$;
```

B Algorithms for Constructing Delta Forwarding Graphs and Checking Invariants

Constructing delta forwarding graphs. Algorithm 4 summarizes the process of constructing delta forwarding graphs. It takes the set of transferred predicates, denoted as D, computed by Algorithm 2, and returns a delta forwarding graph, denoted as $G(V,E,A)$. Here, V is the set of nodes, E is the set of edges, and $A: E \rightarrow 2^{\mathcal{C}}$ is a map from each edge to the set of predicates allowed on that edge ($\mathcal{C}$ denotes the set of all predicates). That is, $A(s1,s2)$ is the set of predicates that can be sent from switch $s1$ to switch $s2$. First, V, E, and A are initialized (Line 1). For each predicate δ in D, APKeep iterates over each *port* in the port set of δ (Lines 2-3). If the node $s1$ holding *port* is not in the node set V, then $s1$ is added into V (Lines 4-6). Similarly, if the node directly connected to *port* is not in V, then $s2$ is also added into V (Lines 7-9). Note here if the *port* is “default”, we assume it is connected to a virtual node named “default”. If the edge $(s1,s2)$ is not in the edge set E, then it is added into E, and the mapping $A(s1,s2)$ is initialized to empty set (Lines 10-12). Finally, the transferred predicate δ is added into the set $A(s1,s2)$ (Line 13).

Checking invariants. With the delta forwarding graph G, operators can check network invariants by traversing G. Algorithm 5 shows an example program for checking blackhole-freedom and loop-freedom (defined in § 3.3). The algorithm starts the traversal from each node $s \in V$. Here V includes all nodes corresponding to the element where the rule is updated. Before each traversal, the algorithm initializes *pset*, the current set of predicates, to D, and *history*, the nodes that have been visited, to empty set (Lines 2-3). The traversal stops when *pset* becomes empty (Line 6-7) or the visited node is already in *history* (Line 8-10), meaning a loop is detected, or the next hop is *default* (Line 12-14), meaning packets in *pset* match no forwarding rule in the corresponding device, i.e., a blackhole is detected. Otherwise, the algorithm updates *pset* and *history* and traverses the next hop (Line 15).

Algorithm 5: `CheckInvariants`(G, D, V)

```
Input: G: the forwarding graph used for invariant checking, D:
       the set of transferred predicates, V: the set of nodes to
       start check.
foreach $s \in V$ do
    $pset \leftarrow D$;
    $history \leftarrow \{\}$;
    Traverse ($s, predicates, history$);
Function Traverse($s, pset, history$):
    if $pset = \emptyset$ then
        return;
    if $s \in history$ then
        Alter('loop');
        return;
    foreach $(s, s') \in E$ do
        if $s' = default$ then
            Alter('backhole');
            return;
        Traverse ($s', pset \wedge A(s, s'), history \cup \{s\}$);
    return;
```

Table 5: Memory cost. TO means timeout (> 24h), and MO means memory overflow (> 32GB).

Network	Memory cost (MB)				
	AP Verifier	VeriFlow	NetPlumber	Delta-net^{MF}	APKeep
Airtel1	417	508	4,283	61	180
Airtel2	5170	16,049	TO	74	193
4Switch	7,722	2,520	1,749	785	936
Internet2	360	206	613	44	87
Stanford*	6	16	4,971	25	3
Purdue*	1,465	1,414	MO	3,414	648
Stanford	6	98	8,607	MO	3
Purdue	TO	TO	MO	MO	744

C Memory Cost

We compare the memory cost of APKeep with the other four methods. Not surprisingly, for single-field datasets, the memory cost of APKeep is comparable to Delta-net^{MF}, and both of them have smaller memory footprint than others. For the multi-field Stanford dataset, Delta-net^{MF} runs out of 32GB memory. For the multi-field Purdue dataset, AP Verifier and VeriFlow do not run to completion within 24 hours; NetPlumber and Delta-net^{MF} run out of 32GB memory. APKeep still maintains a small memory footprint for these two multi-field datasets.

Liveness Verification of Stateful Network Functions

Farnaz Yousefi
Johns Hopkins University

Anubhavnidhi Abhashkumar
University of Wisconsin-Madison

Kausik Subramanian
University of Wisconsin-Madison

Kartik Hans
IIT Delhi[*]

Soudeh Ghorbani
Johns Hopkins University

Aditya Akella
University of Wisconsin-Madison

Abstract

Network verification tools focus almost exclusively on various safety properties such as "reachability" invariants, *e.g.,* is there a path from host A to host B? Thus, they are inapplicable to providing strong correctness guarantees for modern programmable networks that increasingly rely on stateful *network functions*. Correct operations of such networks depend on the validity of a larger set of properties, in particular *liveness* properties. For instance, a stateful firewall that only allows solicited external traffic works correctly if it *eventually* detects and blocks malicious connections, *e.g.,* if it eventually blocks an external host E that tries to reach the internal host I before receiving a request from I.

Alas, verifying liveness properties is computationally expensive and, in some cases, undecidable. Existing verification techniques do not scale to verify such properties. In this work, we provide a compositional programming abstraction, model the programs expressed in this abstraction using compact Boolean formulas, and show that verification of complex properties is fast on these formulas, *e.g.,* for a 100-host network, these formulas result in 8× speedup in the verification of key properties of a UDP flood mitigation function compared to a naive baseline. We also provide a compiler that translates the programs written using our abstraction to P4 programs.

1 Introduction

In recent years, network verification has emerged as a crucial framework to check if networks satisfy important properties. While there are a variety of tools that differ in their focus (*e.g.,* verifying current data plane snapshot vs. verifying a network's control plane under failures), they all share a common attribute: they focus mainly on verifying various flavors of reachability invariants: Is a point in the network reachable from another point [33, 34, 42, 43]? Is there a loop-free path between them [33, 34, 42, 43]? Is the path congested [27,29,38]? Does it traverse a waypoint [41,58]? Is the reachability preserved under link failures or external messages [10, 26]? Are all datacenter shortest paths available for routing [30]? *etc.* Crucially, these tools leave out a richer set of properties that depend on networks guaranteeing *liveness*.

Networks today increasingly deploy complex and stateful network functions such as intrusion detection/prevention systems (IDPS) that monitor traffic for malicious activity and policy violations, and they prevent such activities. To rely on such networks, operators need to verify if "something good (a desired property) eventually happens" [47]—i.e., liveness. As a concrete example, consider a stateful firewall with the policy that a host E external to an enterprise is only allowed to send traffic to an internal host I if I sends a request to E first. The liveness property here is "will a host E that should be allowed to send traffic to I (*i.e., E* was first contacted by I) eventually get whitelisted?", or more precisely that "the event of I sending traffic to E leads to the firewall's whitelisting of E" (§3). Existing reachability-centric tools cannot verify this property: the existence of paths from I to E does not show whether any packet has actually traversed that path; similarly, the reachability of I from E does not give any guarantees whether it was established before or after I sent traffic to E.

Recent work has shown that reachability verification can be made efficient by operating on *Equivalence Classes* (ECs), *i.e.,* groups of packets that experience the same forwarding behavior [33, 34, 43] on a *static* snapshot of the network state. However, verifying liveness is not amenable to such techniques as liveness properties reason about progress and concern the succession of events in dynamic systems. They cannot be verified on a static snapshot of the system.

A dynamic network, in which the state changes, can be modeled as a state machine and, conceptually, existing static verification approaches [35] could be extended to reason about properties of the states and transitions of this machine (§3). However, this naive approach results in state explosion as the network size increases and is therefore impractical (§3, §4).

In this paper, we argue that the goal of verifying liveness properties is achievable using a top-down *function-oriented* strategy that rethinks network abstractions with the efficiency

[*]Work done during an internship at Johns Hopkins University.

of verification in mind. To realize this vision: (a) we provide network programmers with a simple, familiar *abstraction* of one big switch to express their intent. This abstraction enforces the logical separation of different network functions (§2). (b) We then *model* the program as a compact "packet-less" data structure that, unlike the common approach of modeling the forwarding behavior for classes of packets [33,34,42,43], abstracts away the explicit notion of packets and focuses instead on the entities responsible for implementing functions: packet handling rules. (c) We build and evaluate a prototype of our system.

Abstraction: We provide a unified abstraction of one big switch for both control and data planes that conceptually handles all packets. This abstraction closely resembles the simple, familiar data plane abstraction of one big switch directly connecting all hosts [36, 49]. In contrast to the data plane one-big-switch, our abstraction does not require a separate control plane to program it. To reduce verification time, we enforce a *functional decomposition* by requiring the user to implement each function using a separate logical flow table.

Modeling and verification: For verifying properties, similar to prior work [28], we focus exclusively on network behaviors *visible* to users. This enables us to model the system behavior using a compact "packet-less" data structure that abstracts away any details invisible to users, such as the hop-by-hop journey of the packet inside the network [33,34,42,43,52] or even the explicit notion of packets or classes of packets (§3). The packet-less structure models changes in the forwarding behavior as Boolean transitions. We demonstrate the verification efficiency of the packet-less model experimentally (§4), *e.g.,* for a UDP flood mitigation function, within a 1,000*sec.* time-bound, it enables verifying a key liveness property (*host A is eventually blocked*) for networks that are $3.5\times$ larger than those verifiable with the aforementioned naive approach (extending static verification to deal with network states and dynamic transitions).

Implementation and evaluation: We develop a prototype of our design that exposes two interfaces (to express functions on the one-big-switch abstraction and specify verification properties) and a P4 compiler that converts such functions to programs executable on today's programmable devices. Our evaluations show the expressiveness of our interfaces, their low overhead, and the efficiency of our verification design, *e.g.,* for a 100-host network, the packet-less model verifies key liveness properties of a UDP flood mitigation function $8\times$ faster than a packet-based baseline—a gap that increases with the network scale (§4).

2 A Unified Switch Abstraction

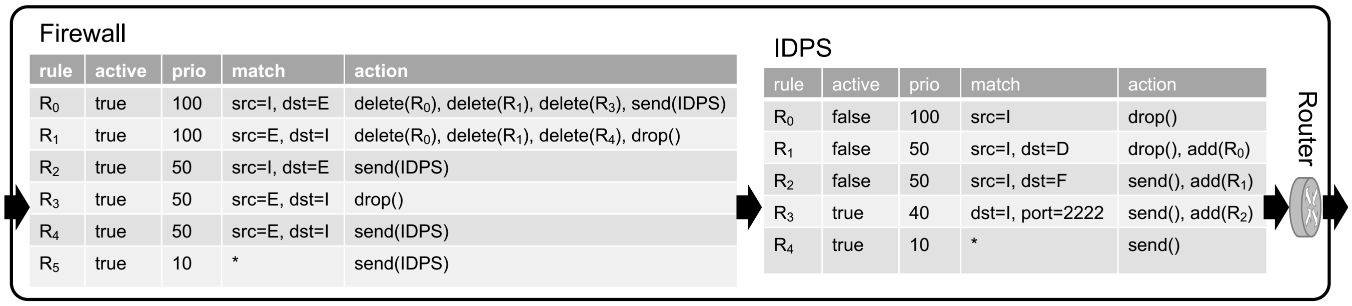
To simplify programming and relieve network programmers of the burden of writing distributed, multi-tier programs, we provide the abstraction of a single, centralized switch that conceptually handles *every* packet. This approach for simplifying programming by having a single unified abstraction for both the control plane and data plane is inspired by Maple [59] and deployed in Flowlog [46]. In contrast to Maple that allows programmers to use standard programming languages and Flowlog's SQL-like language, we start with the most basic and familiar data plane abstraction in networks: one switch that directly connects all hosts. We then augment this abstraction to make it programmable. Similar to Maple and Flowlog, we proactively compile the programs written on our abstraction to control and data plane programs executable in today's networks (§4). We describe our abstraction, its distinction from the common one-big-switch abstraction, and how several canonical network functions can be programmed on it, in turn.

One-big-switch: A common data plane abstraction in networking is that of one logical switch with multiple flow tables, each containing a set of rules, that directly connects all users' hosts together [5, 32, 36, 44]. A rule generally includes: (a) a match field to match against packet headers and ingress ports, (b) priority to determine the matching precedence of rules, (c) counters that are updated when packets are matched, (d) timers that show the time that the rule will be expired and removed from the switch, and (e) actions that are executed when a packet matches the match field of an unexpired rule that has the highest priority among all unexpired rules. These actions could result in changes in the packet, dropping it, or forwarding it. We use *R*.`match`, *R*.`priority`, *R*.`counter`, *R*.`timer`, and *R*.`action` to denote the match, priority, counter, timer, and action of rule *R*.

Our goal is to provide a similar abstraction to this familiar abstraction while making it programmable and amenable to fast verification. In particular, rather than a static, stateless switch, the programmer should be able to implement *dynamic* functions whose behaviors change over time based on traffic. Plus, she should be able to focus solely on the high-level *functionality* that she wishes her switch to provide (*e.g.,* the firewall policies), and the provider of the abstraction is responsible for handling the low-level, distributed implementation of the functionality including reachability (*e.g.,* ensuring that all required packets are correctly forwarded through the firewall). To further assist the user in developing her desired functions, an ideal framework should also provide modular programming and separate the functionality of a program into independent modules. Towards these goals, we make the following alteration in the one-big-switch abstraction [5, 36]:

Add/delete actions: In addition to the standard SDN actions (forward, drop, *etc.*), we allow the execution of a rule to add or remove rules from the switch. As an example, to program a stateful firewall that allows an external host E to talk to an internal host I only after I sends E a request, the switch needs a rule that, upon receiving a request from I to E, alters its state to allow E to I communication.

Actions `add(`*R*`)` and `delete(`*R*`)` show that the execution of the rule results in, respectively, adding and deleting rule



rule	active	prio	match	action
R_0	true	100	src=I, dst=E	delete(R_0), delete(R_1), delete(R_3), send(IDPS)
R_1	true	100	src=E, dst=I	delete(R_0), delete(R_1), delete(R_4), drop()
R_2	true	50	src=I, dst=E	send(IDPS)
R_3	true	50	src=E, dst=I	drop()
R_4	true	50	src=E, dst=I	send(IDPS)
R_5	true	10	*	send(IDPS)

rule	active	prio	match	action
R_0	false	100	src=I	drop()
R_1	false	50	src=I, dst=D	drop(), add(R_0)
R_2	false	50	src=I, dst=F	send(), add(R_1)
R_3	true	40	dst=I, port=2222	send(), add(R_2)
R_4	true	10	*	send()

Figure 1: One-big-switch implements firewall and IDPS.

R. To each rule R, we add a boolean variable R.`active` to show if the rule is currently active on the switch, *i.e.,* packets are matched against it. As an example, the initial state R_0.`active=true`; R_1.`active=false`; R_0.`action=add`(R_1) shows that, unlike R_0, rule R_1 is not initially active and will be activated as a result of R_0's execution.

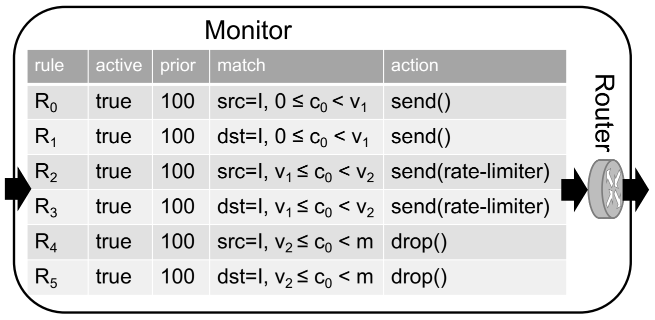
Actionable measurements: For a rule R_i, we allow users to define match fields as predicates not just on packet headers (*e.g., src*=A) but also as conditions in the forms of $l_i \leq c_j < u_i$ on counters, where c_j is the j^{th} counter and l_i and u_i are constants. This condition expresses that for the rule to match a packet, the value of counter c_j should be in the $[l_i, u_i)$ interval.[1] We assume that counters are bounded in the range $[0, m)$. Counters enable the network programmer to easily write critical network functions that depend on traffic statistics such as security applications that detect SYN flood, port scanning, DNS amplification, *etc.* [7, 16, 23] and campus network monitors [35] that block users once their usage exceeds the quota specified by the campus policy. Whenever a rule with a counter is executed on a packet, the counter value is incremented by one. We allow multiple rules to share a counter.

Other optimizations: To improve programmability and verification speed, we also perform the following optimizations: (a) *Functional decomposition:* the user is provided separate tables for each of her network functions, *e.g.,* one table for her firewall, one table for her load balancer, *etc.* If a packet matching a rule R in function F_1 needs to be sent to another function F_2, *e.g.,* an IDPS rule needs to send a packet to the traffic scrubber, this is expressed as R.`action=send`(F_2) in table F_1. (b) *Declarative routing:* Our abstraction provides routing and forwarding as a service, that we call the *router*, to its users. This liberates the user from computing paths and updating them after infrastructure changes, such as changes in policy, topology, and addressing. The user only declares the goal that a packet should reach a destination and delegates the task of figuring out *how* this is done to the provider. Action rules `send`(A) and `send`() simply express the intent of the user to forward a matching packet to the endpoint with ID A and to the packet's destination, respectively. In our prototype, for implementing our declarative routing service, we deploy a basic shortest path forwarding function [1]. (c) *Symbolic rule representation:* the original one-big-switch rules can express a restricted subset of predicates on explicit packet headers, *e.g.,* `src-IP=10.0.0.1` $\wedge$ `dst-IP=10.0.0.2`. Declarative routing enables us to provide a higher-level abstraction to express any general predicate on sets of endpoint identities and header fields, *e.g.,* the programmer can declare a forwarding policy for packets sent to or from a set of hosts called T via a single rule R: R.`match=(src=`T`∨dst=`T`)`, R.`action=send`(), without managing low-level details such as the physical locations and IP addresses of T's hosts.

The changes above turn the data plane one-big-switch abstraction into a unified abstraction that can be efficiently verified: add/delete actions and actionable measurements make the abstraction programmable, and other optimizations accelerate the verification process by reducing the program size (§4). To further assist with efficient verification, our abstraction is designed to have less expressive power than Turing-complete control plane programming languages [46] such as Floodlight [4] and Pyretic [44]. We find experimentally that despite being computationally universal, in practice, control planes perform only a limited set of operations, *e.g.,* adding and deleting rules based on traffic patterns. Our one-big-switch abstraction is designed to be capable of performing similar computations and is therefore expressive enough to program a wide range of network functions. Table 1 lists the functions of a few common control planes and recent network abstractions that we re-wrote on top of our abstraction (implementations in §8). To support the functions that cannot be expressed on top of our abstraction, such as content-based security policies, our framework allows the use of external libraries in conjunction with our abstraction. However, we can only verify the programs expressed on our abstraction.

Examples. Figure 1 shows an illustrative example where the user deploys two tables to implement a chain of two canonical functions: a stateful firewall followed by an IDPS. The first table implements a stateful firewall at the periphery of an enterprise that allows endpoint E (*e.g.,* as an external host) to talk to I (*e.g.,* as an internal host) only if I sends a request to E first. Initially, the table has high priority rules that "watch for" traffic between I and E (R_0 and R_1). If traffic from E to I

[1] If the rule only increases the value of c_j when executed without defining a condition on c_j, we assume l_i=0, and u_i= m, where m is the maximum value supported for counters.



rule	active	prior	match	action
R_0	true	100	src=I, $0 \le c_0 < v_1$	send()
R_1	true	100	dst=I, $0 \le c_0 < v_1$	send()
R_2	true	100	src=I, $v_1 \le c_0 < v_2$	send(rate-limiter)
R_3	true	100	dst=I, $v_1 \le c_0 < v_2$	send(rate-limiter)
R_4	true	100	src=I, $v_2 \le c_0 < m$	drop()
R_5	true	100	dst=I, $v_2 \le c_0 < m$	drop()

Figure 2: One-big-switch implements a monitor.

is observed first, the execution of rule R_1 drops the traffic and changes the state to block E from reaching I. Receiving traffic from I to E first, however, triggers the execution of R_0 which in turn allows bidirectional traffic between I and E to pass through the firewall. Packets not discarded by the firewall are always sent to the next function, the IDPS explained below, for further monitoring. Note that when defining rules in the one-big-switch abstraction, the match fields of the rules must be defined in a way that at least one rule matches each received packet.

The second table implements an IDPS that detects and blocks Trojans. The IDPS determines if the internal host I is infected and needs to be blocked based on a recognizable fingerprint of a backdoor application [16] with the following sequence of operations: (i) I receives a connection on port 2222, (ii) I connects to an FTP server F, and (iii) I tries to connect to the database server D.

Initially, the table contains two active rules: R_3 that matches traffic destined to I on port 2222, and a lower priority rule R_4 that matches all other traffic. Both rules forward the traffic to its destination. The execution of R_3, however, corresponds with the (i) operation above. Once triggered, it activates R_2 that is executed once I tries to reach F (operation (ii)). R_2's execution, in turn, activates R_1, and once I tries to send traffic to D (operation (iii)), it gets blocked (R_1 activates R_0). Once the traffic goes through this pipeline of tables, it is handed to the router to be delivered to its destination.

Figure 2 shows another example for an application that implements a simple campus policy in which an inside-campus host I is allowed to send and receive data before hitting a utilization cap v_1. Once its usage exceeds v_1, but before it reaches v_2, its traffic is routed through a rate limiter. After its usage passes v_2, its access is blocked.[2] A survey of campus network policies shows that universities commonly deploy such usage-based rate-limiting [35]. While for simplicity, we only provide examples of linear chaining of functions in this section (*e.g.,* IDPS after firewall), our abstraction is general enough to express arbitrary dependencies between functions. An example is provided in §8.

[2]In practice, such policies are usually enforced periodically, *e.g.,* the rules (along with their counters set to 0) are re-installed daily.

3 Function Verification

A standard approach for verifying liveness properties of dynamic systems is modeling the behavior of the system as a transition system and expressing its desired properties using temporal logics. The complexity and scalability of this approach depend on the size of the state machine. In this section, we show how we can model stateful dynamic network functions as compact, Boolean "packet-less" transition systems that can be verified efficiently. In §4, we show experimentally that this approach significantly reduces the average verification time for canonical applications compared to a naive packet-based baseline.

3.1 Network as a Transition System

We can model the network as a *transition system*, an analytical framework for reasoning about the behavior of dynamic systems where nodes represent the states of the system (each state corresponds to a valuation of system variables) and edges represent state transitions [8]. Each transition system has a set of initial states as well as a labeling function that maps each node to a set of properties that hold in that state. More formally, a *transition system TS* is a tuple (S, Act, I, AP, L):

- S is a set of states,
- Act is a set of actions,
- $\rightarrow \ \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labeling function.

For convenience, we write $s \xrightarrow{\alpha} s'$ instead of (s, α, s'). Intuitively, a transition system starts in some initial state $s_0 \in I$ and evolves according to the transition relation $\rightarrow$. That is, if s is the current state, then a transition $s \xrightarrow{\alpha} s'$ is selected nondeterministically and taken, *i.e.,* the action α is performed, and the system evolves from state s into state s'.

In networks, the state of the network at each point is its forwarding state (*e.g.,* rules and counters), and transitions are the events that change the state, *e.g.,* policy updates. We next show the properties that can be expressed on these transition systems, explain what makes liveness verification hard, and demonstrate how we can model the network as a compact "packet-less" transition system.

Atomic propositions: *Atomic propositions* are Boolean-valued propositions that express simple known facts about the state of the system. We define these propositions as (h,a) pairs where h and a are, respectively, an equivalence class of packets, and a list of actions (*e.g.,* `send`(I)). For the network transition system TS, an atomic proposition (h,a) holds in a state $s \in S$ if action a applies to all the packets in h.

Labeling function: A *labeling function L* relates a set $L(s) \in 2^{AP}$ of atomic propositions to any state $s \in S$.[3] In the network transition system TS, L maps each state to the set

[3]Recall that 2^{AP} denotes the power set of atomic propositions.

of atomic propositions that hold in that state. That is, (h,a) exists in $L(s)$ if the list of actions a is applied to all the packets in h in state s.

Properties: An execution of a program can be shown as an infinite sequence of states: $s_0, s_1 \ldots$, where each state s_i results from executing a single action in state s_{i-1}.[4] A program's behavior is the set of all such sequences. A *property* is also defined as a set of such sequences [6]. A property *holds* in a program if the set of sequences defined by the program is contained in the property [6]. A partial execution is a finite sequence of program states.

Temporal logic to express properties: Temporal logic is an extension of ordinary logic that facilitates expressing properties via adding assertions about time. Here, we adopt linear temporal logic (LTL) [19], which can express various liveness and safety properties [37,47]. *Temporal logic assertions* are built up from atomic propositions using the ordinary logical operations $\wedge, \vee$, and $\neg$ and some temporal operators—if P and S are atomic propositions: (1) $\mathtt{G}P$ implies "now and forever" P holds, (2) $\mathtt{F}P$ implies "now or sometime in the future" P holds,[5] (3) $P \rightarrow S$ shows logical implication and implies that if P is true now then S will always be true, (4) $P\mathtt{U}S$ implies P remains true "until" S becomes true, and (5) $\mathtt{O}P$ implies P holds in the "next" state. Using the above temporal operators, we can express various properties, *e.g.,* $\mathtt{F}((\mathtt{src}{=}E \wedge \mathtt{dst}{=}I), \mathtt{send}(I))$ [47] asserts that eventually E-to-I packets are delivered, *i.e.,* E can eventually reach I.

3.2 Liveness vs. Safety

A key categorization of properties in distributed systems is into safety and liveness. The categorization is important as the two groups are proved using different techniques [6]. Informally, a *safety* property stipulates that some "bad thing" (deadlocks, two processes executing in critical sections simultaneously, *etc.*) never happens and *liveness* guarantees that "something good" (termination, starvation freedom, guaranteed service, *etc.*) eventually happens [47]. That is, for a property P to be a safety property, if P does not hold for an execution, then at some point, some irremediable "bad thing" must happen. Most of the properties verified in networks today are safety: if a property—such as reachability invariants (E is always reachable from I) [33, 34, 42, 43], waypoint (a certain class of traffic always traverses an intrusion detection system) [41, 58], congestion-freedom [27, 29, 38], and loop-freedom [33, 34, 42, 43]—is violated, then there is an identifiable point—such as a change in the latest snapshot of the network— at which the "bad thing" happens [6].

A partial execution γ is *live* for a property P if and only if there is a sequence of states β such that P holds in $\gamma\beta$. A property for which *all* partial executions are live is a liveness property. In contrast to safety, for liveness properties, no partial execution is irremediable—it is always possible for the required "good thing" to happen in the future [6]. This makes detecting liveness violations challenging as it fundamentally requires an exhaustive search of the entire state space of the network. In the next section, we discuss our approach to overcoming this challenge by modeling networks with compact transition systems. Some examples of liveness properties in networks are (a) the intrusion detection system *eventually* detects all infected hosts, (b) all hosts *eventually* become reachable, *e.g.,* after routing convergence, and (c) showing a recognizable fingerprint of a backdoor application *leads-to* the host being blocked. More generally, "event A leads-to event B" and "event B eventually happens" are two classes of liveness properties [47] as it is always possible for "something good" (*i.e.,* event B) to happen. An example of a property that includes liveness is *total correctness* which is composed of partial correctness (the program never generates an incorrect output; a safety property) and termination (the program generates an output; a liveness property) [6].

3.3 Packet-less Model

Feasibility of model checking is tied inherently to handling state explosion [19]. To mitigate this problem, we try to provide a compact "packet-less" structure that models only the entities that perform the functions in the network: packet handling rules. Plus, we model rules in the most abstract form: as *Boolean* variables, abstracting away all of the attributes of rules, such as their match fields, actions, and priorities. This is in contrast to pervasive network verification techniques that model the network behavior in terms of packets and equivalence classes of packets (ECs) [34, 35, 43].

Boolean variables and formulas provide a more compact way to represent the state space of a transition system. State-of-the-art model checkers like NuSMV [18] use symbolic techniques, such as Binary Decision Diagrams (BDDs), for efficiently exploring transition systems that have a Boolean representation. Such representations enable model checkers to explore extremely large state spaces efficiently [15]. We next explain how we can encode the behavior of dynamic network functions as packet-less models. We then show in §4 that this approach of packet-less modeling results in a significant reduction in the verification time of canonical network functions compared to a packet-based approach.

Compact, packet-less models: We initially model the network as a transition system TS with a single Boolean variable corresponding to each rule. This variable is true if the rule is active in the network and false otherwise. As a starting point, we abstract away counters, assuming that rules do not depend on them. (We will later refine this model to incorporate counter semantics.) Each node in this structure shows one **state** of the network defined by a valuation of Boolean

[4]Terminating executions are modeled as sequences where the last state reached by the terminating trace repeats infinitely.

[5]Note that $\mathtt{F}$ is the dual of $\mathtt{G}$, *i.e.,* $\mathtt{F}P$ is equivalent to $\neg\mathtt{G}\neg P$, where P is an atomic proposition [47].

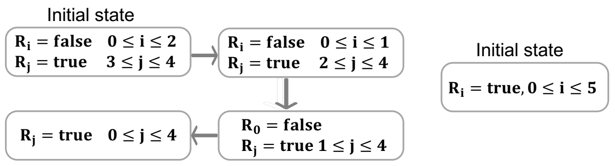

Initial state
R_i = false $0 \le i \le 2$
R_j = true $3 \le j \le 4$
R_i = false $0 \le i \le 1$
R_j = true $2 \le j \le 4$
R_0 = false
R_j = true $1 \le j \le 4$
R_j = true $0 \le j \le 4$
Initial state
R_i = true, $0 \le i \le 5$

Figure 3: Packet-less models of the (a) IDPS and (b) monitor.

rule variables. S is the set of all these states. A rule can be *executed* in a state if, for at least one packet that it matches, it is the highest priority rule with value=`true` in that state. **Actions** are the execution of rules that update the forwarding state by adding or deleting rules, and **transitions** show how the state evolves as a result of these actions. If in a network state $s \in S$, rule R_i can be executed and its execution adds or deletes rules, then there is a transition $s \ \xrightarrow{R_i} \ s', s' \in S$, where for each added rule R_j, R_j=`true` in s' and for each deleted rule R_k, R_k=`false` in s'. In the initial state, the value of each R_i is equal to the initial value of R_i.`active` in the original program. Figure 3 shows the packet-less models for the IDPS and the campus monitor functions in Figures 1 and 2 (not including labeling functions).

Despite being more efficient, not explicitly modeling packets causes some key challenges: (1) Abstracting away the notion of packets makes it challenging to incorporate the semantics of traffic statistics such as counters. (2) As §3.1 explains, properties are defined on packets and whether a property holds in a network state depends on the actions of the highest priority rule that matches the packet. Thus, it is challenging to verify properties on a structure that abstracts away any explicit notion of packets, headers, priorities, *etc.* For the IDPS function in Figure 1, for example, the packet-less model has two rules in the initial state that can match packets sent to I with port number 2222, as Figure 3 (a) shows. With each rule modeled as a Boolean variable, it is not possible to determine which matching rule handles a packet (and therefore verify properties). We address these challenges next.

3.3.1 Boolean Formulas of Counters

The first challenge of the Boolean packet-less modeling is preserving the semantics of stateful programs with traffic counters. Recall that in packet-less models, we initially abstracted away counters. We next show how to refine these models to incorporate semantics of counters.

Refining states: To model counters, we observe that if the only variable that changes across a set of network states is a counter value and the value of this counter does not pass counter conditions, then the forwarding behavior remains the same in all those states. In the monitor function, the network behavior is identical for all counter values between 0 and v_1. This allows us to track counter *predicates*, Boolean-valued functions on counters, instead of actual counter values. In the monitor function, we can define the following three predicates: $(0 \le c_0 < v_1)$, $(v_1 \le c_0 < v_2)$, and $(v_2 \le c_0 < m)$. In the initial state, only the first predicate, $(0 \le c_0 < v_1)$, is true.

The fact that the forwarding behavior is determined not by exact counter values but by counters passing thresholds makes counters amenable to *predicate abstraction*, a powerful technique to mitigate the challenges of verifying programs with large base types such as integers [19]. This technique reduces the size of the model by tracking only predicates on data and eliminating invisible data variables.

Concretely, counter conditions partition an interval into subintervals that may have distinct forwarding behaviors. Let R_i be a rule that depends on the j^{th} counter, c_j, *i.e.,* it is active if c_j's value is in the $[l_i, u_i)$ range, and $P_j = \text{order}(\cup_i(l_i \cup u_i))$, *i.e.,* an ordered list (in non-decreasing order) of all lower and upper bounds of all rules that depend on c_j. $P_{j,k}$, $l_{j,k}$, and $u_{j,k}$ denote, respectively, the k^{th} subinterval of counter c_j, its lower bound, and its upper bound. In the monitor program, $P_{0,0} = [0, v_1)$ is the first subinterval of the first counter, $l_{0,0} = 0$, and $u_{0,0} = v_1$. When the counter value is in this subinterval, rule R_0 can handle packets as its counter conditions, $(l_0 \le c_0 < u_0)$, are satisfied, *i.e.,* $(l_0 \le l_{0,0}) \wedge (u_{0,0} \le u_0)$ where $l_0 = 0$ and $u_0 = v_1$. R_2, on the other hand, cannot handle packets because its counter conditions are not satisfied in this subinterval, *i.e.,* $(l_2 \not\le l_{0,0}) \wedge (u_{0,0} \not\le u_2)$ where $l_2 = v_1$ and $u_2 = v_2$.

We refine TS by adding a Boolean variable R'_i for every rule R_i. Our goal is to set the value of this variable to `true` in a state s if the counter conditions of R_i are satisfied in s and to `false` otherwise. For any rule R_k that does not depend on counters, R'_k=`true`.[6] The numbers in P_j are the only places where R'_i variables of the rules that depend on c_j can change. In the monitor program, $P_0 = [0, v_1, v_2, m]$ lists the only points in the $[0, m)$ range where the condition of a rule that depends on counter c_0 such as R_0 can transition from `true` to `false` and vice versa.

For each counter c_j in a state in the packet-less model, we partition the state into $|P_j| - 1$ states, where $|P_j|$ is the number of points in P_j, *e.g.,* P_0=4 in the monitor example. Each of these $|P_j| - 1$ states corresponds to one subinterval. Suppose that R_i is a rule that depends on counter c_j, *i.e.,* R_i can handle packets when the value of the counter satisfies its counter conditions: $l_i \le c_j < u_i$. The value of R'_i in each refined state should show whether the counter conditions of rule R_i are satisfied in the corresponding subinterval $P_{j,k}$: $R'_i = ((l_i \le l_{j,k}) \wedge (u_{j,k} \le u_i))$.

In any given state $s \in S$ and for a subinterval of $P_{j,k}$, the network behavior is determined by the rules that (a) are active in that state (*i.e.,* R_i=`true`) and (b) either do not depend on counters or their counter conditions are satisfied in that subinterval (*i.e.,* R'_i=`true`), *e.g.,* in the monitor example's initial state, R_0 and R_1 are active (*i.e.,* R_0=R_1=`true`) and their counter conditions are satisfied in the first subinterval $P_{0,0} = [0, v_1)$ (*i.e.,*

[6]Note that it is possible to avoid defining these variables for the rules that do not depend on counters. We define these variables for all rules here for ease of exposition.

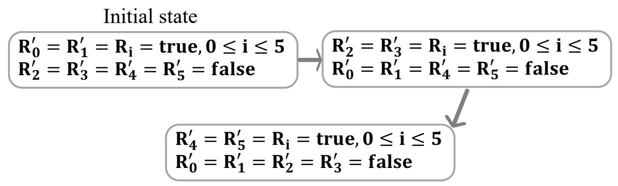

Initial state
$R'_0 = R'_1 = R_i = \mathbf{true}, 0 \le i \le 5$
$R'_2 = R'_3 = R'_4 = R'_5 = \mathbf{false}$
$R'_2 = R'_3 = R_i = \mathbf{true}, 0 \le i \le 5$
$R'_0 = R'_1 = R'_4 = R'_5 = \mathbf{false}$
$R'_4 = R'_5 = R_i = \mathbf{true}, 0 \le i \le 5$
$R'_0 = R'_1 = R'_2 = R'_3 = \mathbf{false}$

Figure 4: The refined structure for the monitor function.

R'_0=R'_1=`true`).

In summary, the set of variables in the packet-less model TS includes a pair of Boolean variables R_i and R'_i for each rule. The values of these variables at any point define the **state** of the packet-less model at that point. In the **initial state**, for every rule R_i, the value of R_i is equal to R_i.`true` in the original program. In the initial state, R'_i=`true` *iff* R_i either does not depend on any counters or if it depends on counter c_j and its counter conditions are satisfied in the first subinterval of c_j, $P_{j,0}$, *i.e.,* $((l_i \le l_{j,0}) \wedge (u_{j,0} \le u_i))$.

In the refined model, a rule R_i can be *executed* in a state if (a) for at least one packet that R_i matches, it is the highest priority rule, (b) R_i is active in that state (*i.e.,* R_i=`true`), and (c) R_i either does not depend on any counters or its counter conditions are satisfied in that subinterval (*i.e.,* R'_i=`true`).

Transitions in the refined model: Let s_k and s_{k+1} be, respectively, the k^{th} and $(k+1)^{\text{th}}$ refined state if state s is refined for counter c_j, *i.e.,* the states representing the Boolean formulas for subintervals k and $k+1$ of P_j. We add a transition from state s_k to state s_{k+1} if there is at least one rule R_i that depends on c_j and can be executed in s_k. The reason is that c_j increases monotonically so if a counter-dependent rule can be executed in s_k, it can result in c_j transitioning to the next subinterval (corresponding to state s_{k+1}). For each transition $s \rightarrow^{\alpha} s'$ in the original packet-less model before refinement, where α is the execution of a rule R_i that adds or deletes rules, there is a transition $s_k \rightarrow^{\alpha} s'_k$ if R_i can be executed in s_k. As before, for each added rule R_j that R_i adds, R_j=`true` in s'_k and for each rule R_k that R_i deletes, R_k=`false` in s'_k. Figure 4 shows the Boolean packet-less states and the transitions for the monitor function in Figure 2 (see Figure 3 (b) for its corresponding pre-refinement packet-less model). There is a transition from the initial partitioned state (in which $R'_0 = R'_1$=`true`) to a state in which $R'_0 = R'_1$=`false`, reflecting the fact that the network state evolves from an initial state that satisfies the counter conditions of these rules to one where the counter conditions of these rules are no longer satisfied. If the program has multiple counters, the state and transition refinements are performed sequentially for each counter.

3.3.2 Boolean Formulas of Properties

We explained earlier how we could express *atomic propositions* in terms of packets and the desired actions on them. Properties are built out of atomic propositions, and in a transition system such as the packet-based model in prior work [35], the atomic proposition (h,a) holds for a state *iff* the list of actions a is applied to all the packets in h in that state. The second challenge of packet-less modeling is evaluating a proposition on Boolean packet-less models. In this part, we explain how an (h,a) proposition can be expressed as a Boolean formula on rules exploiting a priori known rule priorities.

We say that a rule R_i can be *applied* if (a) R_i is active, *i.e.,* R_i=`true` and (b) R_i either does not depend on any counters or its counter conditions are satisfied, *i.e.,* R'_i=`true`. For a rule to be executed, in addition to satisfying the conditions above, for at least one packet that it matches, it should be the highest priority rule.

Let $W_{0,n} = [R_0, R_1, ..., R_n]$ and $W'_{0,n} = [R'_0, R'_1, ..., R'_n]$ be, respectively, the list of rules R_i and the list of variables R'_i, sorted in non-increasing order of the priorities of rules, *i.e.,* R_i.`priority` $\ge R_j$.`priority`, R'_i.`priority` $\ge R'_j$.`priority` if $i < j$. Our goal is to express whether a proposition holds in a state as a Boolean formula on this list. We achieve this with a recursive function: Let $K((h,a), W_{0,n}, W'_{0,n})$ be a function that is true if $W_{0,n}$ and $W'_{0,n}$ satisfy the proposition (h,a) and false otherwise. For the two special cases, where (1) h is empty and (2) h is not empty and the list of rules is empty, we assume that $K((h,a), W_{0,n}, W'_{0,n})$ evaluates to, respectively, `true` and `false` because any condition holds for a non-existing packet (item (1) above, $h = \varnothing$) and an empty set of rules does not satisfy any conditions for packets (item (2) above, $h \neq \varnothing$ and the list of rules=[]). In other conditions, we have the following cases:

Case 1: If the highest priority rule R_0 matches some packets in h, *i.e.,* if $(h \cap R_0.\texttt{pkts}) \neq \varnothing$, where $R_i.\texttt{pkts}$ denotes the set of packets that R_i matches, and its action includes the actions in the proposition, *i.e.,* $a \subset R_0.\texttt{action}$, then for the proposition to hold, one of these two conditions should hold (a) either R_0 can be applied ($R_0 \wedge R'_0$) and for all the packets of h that do not match R_0, the proposition should hold for the next, lower-priority matching rules, *i.e.,* $(R_0 \wedge R'_0) \wedge (K((h - R_0.\texttt{pkts}, a), W_{1,n}, W'_{1,n}))$, or (b) R_0 cannot be applied ($\neg(R_0 \wedge R'_0)$, R_0 either is not installed in the network or its counter conditions are not satisfied), but in this case, for all the packets of h, the proposition should hold for the next, lower-priority matching rules, *i.e.,* $\neg(R_0 \wedge R'_0) \wedge (K((h,a), W_{1,n}, W'_{1,n}))$. In other words, $K((h,a), W_{0,n}, W'_{0,n}) = ((R_0 \wedge R'_0) \wedge (K((h - R_0.\texttt{pkts}, a), W_{1,n}, W'_{1,n}))) \vee (\neg(R_0 \wedge R'_0) \wedge (K((h,a), W_{1,n}, W'_{1,n})))$.

Case 2: If the highest priority rule R_0 matches some packets in h, *i.e.,* if $(h \cap R_0.\texttt{pkts}) \neq \varnothing$, and its action does not include the actions in the proposition, then for the proposition to hold, it should not be possible to apply R_0, *i.e.,* $\neg(R_0 \wedge R'_0)$. Otherwise, it matches the packets but does not apply the intended actions on them. Plus, for all the packets of h, the proposition should hold for the next, lower-priority matching rules, *i.e.,* $K((h,a), W_{0,n}, W'_{0,n}) = \neg(R_0 \wedge$

$R'_0) \wedge (K((h,a),W_{1,n},W'_{1,n}))$.

Case 3: If the highest priority rule R_0 does not match any packets in h, then irrespective of whether R_0 can be applied or not, for all the packets of h, the proposition should hold for the next, lower-priority matching rules, *i.e.,* $K((h,a),W_{0,n},W'_{0,n}) = K((h,a),W_{1,n},W'_{1,n})$.

Applied recursively to the ordered list (according to priority) of all rules in the network, $K((h,a),W_{0,n},W'_{0,n})$ expresses the proposition (h,a) as a Boolean formula on R_i and R_i' variables. As an example, in the IDPS function of Figure 1, the proposition $((\texttt{src=}I,\texttt{dst=}E),\texttt{send()})$ is translated to

$$\begin{aligned}
K(((\texttt{src=}I,\texttt{dst=}E),\texttt{send()}),W_{0,4},W'_{0,4}) &= \\
\neg(R_0 \wedge R'_0) \wedge (K(((\texttt{src=}I,\texttt{dst=}E),\texttt{send()}),W_{1,4},W'_{1,4})) &= \\
\neg(R_0 \wedge R'_0) \wedge K(((\texttt{src=}I,\texttt{dst=}E),\texttt{send()}),W_{2,4},W'_{2,4}) &= \\
\neg(R_0 \wedge R'_0) \wedge K(((\texttt{src=}I,\texttt{dst=}E),\texttt{send()}),W_{3,4},W'_{3,4}) &= \\
\neg(R_0 \wedge R'_0) \wedge K(((\texttt{src=}I,\texttt{dst=}E),\texttt{send()}),W_{4,4},W'_{4,4}) &= \\
\neg(R_0 \wedge R'_0) \wedge & \\
(\,(\,(R_4 \wedge R'_4) \wedge K((\varnothing,\texttt{send()}),[])) \vee & \\
(\neg(R_4 \wedge R'_4) \wedge K(((\texttt{src=}I,\texttt{dst=}E),\texttt{send()}),[]))) &= \\
\neg(R_0 \wedge R'_0) \wedge (R_4 \wedge R'_4). &
\end{aligned}$$

This Boolean formula results from applying the rules specified in cases (2), (3), (3), (3), (1), and the first two special cases, respectively. It has different truth values in different states in Figure 4 depending on the truth values of rules in those states, *e.g.,* it is true in the initial state S_0 where R_0=`false` and R_4=`true`, meaning that the assertion $((\texttt{src=}I,\texttt{dst=}E),\texttt{send()})$ holds (I may talk to E in this state), but is false in the final state (I is blocked), where R_0=`true` and R_4=`true`. Note that R'_i variables are always `true` in this example as the rules do not depend on counters.

In the network transition system TS, **labeling function** L maps each state to the set of atomic propositions that hold in that state. That is, (h,a) exists in $L(s)$ if the list of actions a is applied to all the packets in h in state s, *i.e.,* $K((h,a),W_{0,n},W'_{0,n})$=`true` where $W_{0,n}$ and $W'_{0,n}$ are, respectively, the list of rules R_i and the list of variables R'_i (sorted in non-increasing order of rule priorities) in s.

4 Implementation and Evaluation

To evaluate the performance of our design, we build a prototype that enables network operators to program and verify their functions and a compiler that converts these functions to programs executable on programmable switching ASICs. After a brief overview of our prototype, we show that our network abstraction and specification language are expressive and impose only minimal overhead. We also show that compared to a packet-based baseline, the packet-less model's verification of different properties is faster and more scalable, *e.g.,* for a network with 100 hosts, the packet-less model results in 8$\times$ speedup in the verification of liveness properties of a UDP flood mitigation function compared to the packet-based model.

4.1 Implementation

Interfaces: Our system exposes two interfaces: a *one-big-switch* interface that enables a network operator to program her functions on our abstraction (§2) and a *specification* interface that allows her to express her desired properties (§3). Our *generator* then automatically builds the packet-less model and the Boolean formula representing the specification as explained in §3 and interacts with NuSMV, a state-of-the-art model checker [18], to verify specification properties.

Compiling the abstraction to P4 programs: P4 [14] is a language for expressing the packet processing of programmable data planes. Along with the programmable data plane, the control plane is responsible for populating the tables defined by the P4 program. We build a compiler to compile a one-big-switch program using our abstraction to a P4_16 program for the P4 behavioral model [3], an open-source programmable software switch. Along with the software switch, we develop a control plane which adds and deletes table rules. We describe some of the salient features of the compilation:

(a) Functional decomposition: We map each table in the abstraction to a P4 table, whose match fields and actions are constructed using the rules in the table. Network function traversal policies are implemented using P4 control flow constructs, *e.g.,* to traverse a firewall table `fw` conditionally:

```
    if (meta.visit_fw) == 1)
        fw.apply();
```

where `meta.visit_fw` is a metadata variable used by the tables before the firewall table to ensure the packet goes through the firewall table.

(b) Actionable measurement: We use P4's registers to support incrementing and matching on counters shared across multiple rules. If a table in the abstraction uses counters, we create a separate P4 table which is responsible for updating the shared counters and transferring the counter state to metadata such that the function table can use the value for matching. This helps us confine register accesses to a single table. Thus, packet processing can happen at the line rate [54].

(c) Add/delete actions: Currently, P4 data planes do not support add/delete actions, *i.e.,* rule actions that add or delete other table rules, due to hardware limitations of existing platforms. We support this functionality by cloning the packet [55] to the control plane, acting similar to the PacketIn functionality in OpenFlow [2]. When a rule in the program has an add/delete action, our switch program clones the packet in the data plane and sends it to the switch-local

Function source	Function	One-big-switch LoC	Packets calling controller [%]	P4 LoC	Compilation time[ms]
Pyretic [44]/Kinetic [35]	Simple counter	1	0	144	1.8
	Port knocking	3	6×10^{-3}	133	1.8
	Simple firewall	2	0	128	1.6
	IP rewrite	2	0	134	1.8
	Simple rate limiter	3	0	141	2.1
Floodlight [4]	Firewall/ACL	2	0	132	1.7
Chimera [13]	Phishing/Spam detection	3	6×10^{-3}	151	2.3
FAST [45]	Simple stateful firewall	3	0.3	133	2.1
	FTP monitor	2	0	135	2.1
	Heavy-hitter detection	2	0	148	2.0
	Super spreader detection	2	0	148	2.0
	Sampling based on flowsize	6	0	189	2.5
Bohatei [23]	Elephant flow detection	3	0	182	2.4
	DNS amplification mitigation	3	7×10^{-3}	135	1.9
	UDP flood mitigation	2	0	148	1.8

Table 1: Applications written on one-big-switch.

control plane; the control plane (which we also generate) then adds/deletes table rules as decided by the add/delete actions in the program. This approach has the overhead of punting some packets (specifically, first packets in a connection matching specific rules) to the local switch CPU—this is a cost we pay for lack of data plane support for add/delete actions.

4.2 Evaluation

Expressiveness: Despite its simplicity, our one-big-switch abstraction enables developers to express a broad range of applications and network functions. Table 1 shows a list of functions that we have developed in our framework. Full descriptions of these programs are provided in §8. Network policies can be succinctly expressed on our one-big-switch abstraction in only a few lines of code (column 3 in Table 1), *e.g.,* a simple stateful firewall policy that allows only the traffic whose connection was initiated by a host in a given department can be expressed in 3 lines of code on the one-big-switch abstraction. The compiled P4 code of the same policy is expressed in 133 lines of code (comprising of 100 lines of boilerplate code for headers and parsers) and ~50 lines of code for the P4 control plane for cloning the packets at the data plane and adding rules from the control plane. Various specification properties, including the most common specification patterns in practice [21], can also be specified in the temporal logic presented in §3.

Limited overhead: Add/delete actions of our abstraction are not directly supported in the switching ASIC today. Our P4 compiler implements these by involving the controller whenever a rule has such actions. To measure the overhead of involving the controller, we deploy the functions listed in Table 1 and replay a packet trace of a university datacenter [12], with over 102K packets and 1,791 IP addresses and measure the frequency of calling the controller. This overhead is modest for all functions, *i.e.,* 0-0.3% (column 4, Table 1).

Verification time: We test the efficiency of bounded-time verification of packet-less and packet-based models at scale to answer questions such as, what functions and properties are verifiable with each approach? How does the verification time scale with respect to the network (and hence the model) size? How does it scale with respect to the property size? To do so, for the functions in Table 1 with per-host policies (*e.g.,* the heavy-hitter detector, port knocking, rate limiter, phishing/spam detector, and UDP flood mitigation function), we define one policy for each host in the network. The heavy hitter detector, for example, deploys a per-source IP counter that is incremented for every new SYN packet and starts dropping packets when the counter exceeds a threshold (Table 4). For the functions that define policies between communicating pairs of hosts or on flows, *e.g.,* FTP monitor and DNS amplification mitigation functions, we run the same datacenter trace as above to find the communicating pairs and matching flows and define one policy for each. For functions that classify hosts into sets, *e.g.,* the firewall function with sets of internal and external hosts, we randomly assign each host to a set. Finally, for counter bounds, we draw random samples from the uniform distribution on the set of possible values.

We measure the verification time for various functions, properties, and network scales for packet-less and packet-based models. Note that increasing the network size in this manner results in larger models. For each experiment, we run 20 repeated trials, each trial with a time budget of 1,000 *sec.*, *i.e.,* we stop the verification process when the verification time exceeds 1,000 *sec*. We give a brief overview of our packet-based baseline below before presenting our results.

Packet-based transition systems as the baseline: A powerful technique for scaling static verification is slicing the network into a set of *equivalence classes* (ECs) of packets [33,34,43]. Each EC is a set of packets that always experience the same forwarding actions throughout the network [43]. As shown in prior work, this approach can be extended to model dynamic networks [35]. To do so, one can detect a program's ECs using verifiers that classify packets into ECs [43].

Rule	Init	Priority	Match	Action
R_0	true	100	(protocol=UDP) & (source IP=A) & ($c_0 < X$)	send()
R_1	true	100	(protocol=UDP) & (source IP=A) & ($X \leq c_0 < m$)	drop()

Table 2: A UDP flood mitigation function.

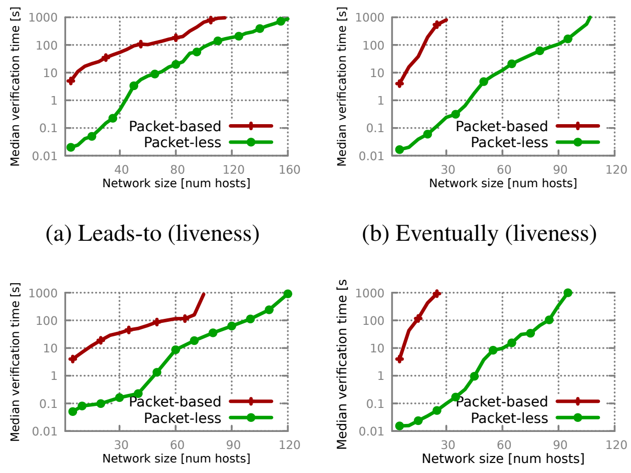
Alternatively, and similar to Kinetic [35], the programmer may be tasked with providing ECs and their transition systems.

In the packet-based transition system, each *node* represents a *state* of the network, *i.e.,* the set of ECs in that state. For instance, for the IDPS in Figure 1, in the initial state, two ECs exist in the network: EC_0 that includes all packets destined to I and with port number 2222 and EC_1 that includes all remaining packets. *Transitions* are the events that change the state of the network, *e.g.,* receiving packets from the ECs whose forwarding actions update the network's forwarding behavior. In the example above, receiving packets from EC_0 transitions the system into another state in which three different ECs exist: EC_2 that includes all packets from I that are destined to F, EC_3 that includes all packets not included in EC_2 that are sent to I with port number 2222, and EC_4 that includes all packets not included in EC_2 and EC_3.

We implement this packet-based model as our baseline. For classifying packets into ECs in this baseline, we use the heuristic developed in VeriFlow [43]. Despite their similarities, the packet-based model and Kinetic [35] have a few key differences: the greater expressiveness of our programming abstraction (*e.g.,* to allow for matching on shared packet counters) increases the difficulty of the verification problem. Plus, in contrast to Kinetic that requires operators to provide the state machine as an input, the packet-based model automatically generates these from the rules written on our one-big-switch abstraction.[7] Thus, despite their conceptual similarities, we refrain from calling our baseline Kinetic.

Our results demonstrate that the packet-less model is faster than the packet-based model for different categories of liveness properties such as "leads-to" (*e.g.,* host A sending traffic to host B leads to A being blocked) and "eventually" (*e.g.,* host A is eventually blocked). Figure 5 (a) and (b) show examples for a UDP flood mitigation function (Table 2) which deploys a per-source IP counter that is incremented for every UDP packet and starts dropping packets when the counter exceeds a threshold. Verifying a leads-to property—host A sending more packets than a threshold leads to it being blocked—in a network with 100 hosts, for example, is 7.8× faster with the packet-less model than the packet-based model (85 vs. 667 *sec.*). In 1,000 *sec.*, the packet-less model verifies a different liveness property, "eventually" (A is eventually blocked), for networks that are 3.5× larger than those verifiable with the packet-based model (105 hosts *vs.* 30). By reducing the size of the state machine, the packet-less model also improves the ver-

[7]In Kinetic, a knowledgeable operator can conceivably provide compact state machines, smaller than the packet-based model, and hence experience lower verification times.



(a) Leads-to (liveness)

(b) Eventually (liveness)

(c) Absence (safety)

(d) Universality (safety)

Figure 5: Packet-less verification reduces the verification time of different properties for a flood mitigation function.

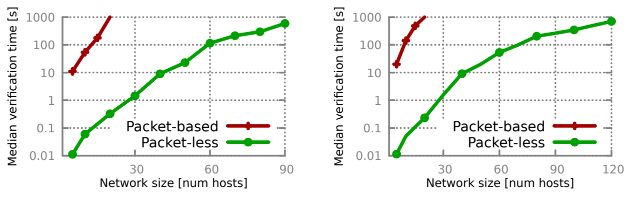
ification time for safety properties. Figure 5 (c) and (d) show examples for an "absence" property (host A is never reachable) and "universality" (A is always reachable), respectively. Similarly, verifying the same liveness and safety properties, *e.g.,* "eventually" and "universality", in a stateful firewall (Table 13) is, respectively, 3.3× and 4.9× faster with the packet-less model than the packet-based model in a 30-host network (figures not shown).

We observe that greater degrees of state sharing across rules (*e.g.,* counters shared by multiple rules) result in slower verification for both approaches, but the performance degradation is more pronounced for the packet-based model, *e.g.,* for a rate limiter (Table 5), in 1,000 *sec.*, we can verify an "eventually" property in networks with up to 90 hosts with the packet-less model (*v.s.* 105-host networks for the UDP flood detection application above), and for networks with at most 15 hosts with the packet-based model (*v.s.* 30 hosts for the UDP flood detection application above). Figure 6 shows the results for a liveness and a safety property for this function. The packet-less model can verify them for network 6× larger than the packet-based model, and even for small-scale networks, it is at least two orders of magnitude faster.

In addition to testing the scalability with respect to the network (and consequently the model) size, we also scale properties. Verification time is a function of the size of the transition system and the property formula, *e.g.,* there exists an LTL model-checking algorithm whose running time depends linearly on the size of the transition system and exponentially

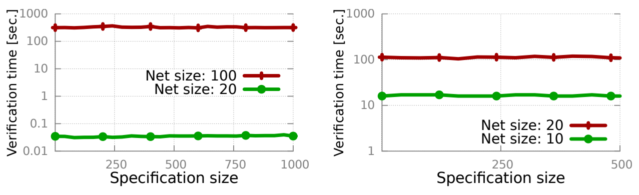
Property	LTL specification [17,21]	Meaning and example
Leads-to (a.k.a Response)	$\mathtt{G}(P \rightarrow \mathtt{F}S)$	*P* must always be followed by *S*, a host showing a malicious activity must always be followed by the IDPS blocking the host.
Universality	$\mathtt{G}P$	*P* always holds, *e.g., A* is always blocked.
Absence	$\mathtt{G}\neg P$	*P* never holds, *e.g., A* can never send traffic to *B*.
Eventually (a.k.a Existence)	$\mathtt{F}P$	*P* eventually happens, *e.g., A* can eventually reach *B*.
Precedence	$\mathtt{F}P \rightarrow (\neg P \mathtt{U}(S \wedge \neg P))$	*P* must always be proceeded by *S*, *e.g.,* blocking a host must always be proceeded by the host exhibiting some malicious activity.

Table 3: List of properties verified.



(a) Eventually (b) Universality

Figure 6: Verification times for a rate limiter.



(a) Packet-less (b) Packet-based

Figure 7: The property size does not impact the verification time.

on the length of the LTL formula [20]. Although in practice, the structure size is usually the dominant factor in determining the verification time, in our approach, the size of property formulas can potentially be large and increase the verification time.[8] To test the sensitivity of the verification time to the property size, we first construct atomic propositions for reachability of randomly selected (with replacement) hosts, *e.g.,* ($\mathtt{dst}$=A,$\mathtt{send}(A)$). Then, we use logical operations to build formulas of various sizes out of these atomic propositions, as explained in §3, *e.g.,* $(\mathtt{dst}{=}A,\mathtt{send}(A))\wedge(\mathtt{dst}{=}B,\mathtt{send}(B))$ for a property with size 2. Finally, we use these assertions to construct the liveness and safety properties listed in Table 3. We observe that scaling the property size does not affect the verification time of either approach. For the same UDP flood mitigation function as above, in a network with 200 hosts, changing the property size from 1 to 200, results in a standard deviation of less than 7.2. This holds even for smaller networks where the relative impact of the property size is expected to be greater. Figure 7 shows two examples for verifying $\mathtt{F}(\wedge_{h\in S}(\mathtt{dst}{=}h,\mathtt{send}(h))$, *i.e.,* eventually, all the hosts in the set *S* will become reachable, for small networks with 10, 20, and 100 hosts.

[8]Recall that in a packet-less model, we also need to express properties in terms of Boolean formulas on rules (§3.3.2).

5 Related Work

Static network verifiers [33,34,39,42,43,61] verify various aspects of reachability invariants such as loop-freedom and lack of blackholes on a snapshot of the network. More recently, reachability verification and enforcement techniques are extended to incorporate degrees of dynamism [10,22,26,40,48, 60], *e.g.,* with failures and policy changes [26,27,29,31,38, 52], with mutable data planes [48], and with focusing on controllers instead of snapshots of the data plane [10,11,22,60]. However, the targeted properties in all these proposals are safety invariants such as reachability and loop-freedom. Our focus on verifying a computationally more complex category of properties (liveness) drives our novel packet-less model, which is distinct from prior models. Plus, some of the assumptions in prior work restrict the set of network functions that they can verify. VMN, for example, models network functions in which (a) flows are independent, and (b) forwarding is not affected by transaction orderings. We find that many crucial network functions such as IDPS and Trojans detectors [16] do not possess those properties.

We share the goal of designing verifiable programming languages with VeriCon [9], FlowLog [46], and Kinetic [35], but since our programs are compiled to P4 programs (instead of the OpenFlow rules that VeriCon's CSDN language, FlowLog, and Kinetic programs are compiled to), we are able to express programs that these frameworks cannot, *e.g.,* programs with multiple rules that share counters. Plus, VeriCon's use of first-order logic makes it infeasible to specify dynamic properties such as liveness. Finally, VeriCon, FlowLog, and Kinetic are packet-based. Kinetic, for example, extends the Pyretic controller [44] to add support for the verification of dynamic networks based on packet equivalent classes [35]. In Kinetic, the programmer needs to specify "located packet equivalence classes" (LPECs), maximal regions of the flow space (*e.g.,* packets with a given source IP) that experience the same forwarding behavior in each state, and their associated finite state machines (FSMs) that encode the handling of LPECs. We show experimentally and theoretically that compared to packet-based approaches deployed in these proposals, our packet-less approach results in faster verification.

Many recent projects in network verification leverage classical concepts of model checking to control the state explo-

Rule	Init	Priority	Match	Action
R_0	true	100	(TCP flag=SYN) & (source IP=A) & $(0 \leq c_0 < X)$	send()
R_1	true	100	(TCP flag=SYN) & (source IP=A) & $(X \leq c_0 < m)$	drop()

Table 4: A heavy-hitter detection function.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & $(0 \leq c_0 < v_1)$	send(port=1)
R_1	true	100	(source IP=A) & $(v_1 \leq c_0 > v_2)$	send(port=2)
R_2	true	100	(source IP=A) & $(v_2 \leq c_0 < v_3)$	send(port=3)
R_3	true	100	(source IP=A) & $(v_3 \leq c_0 < v_4)$	send(port=4)
R_4	true	100	(source IP=A) & $(v_4 \leq c_0 < v_5)$	send(port=5)
R_5	true	100	(source IP=A) & $(v_5 \leq c_0 < m)$	drop()

Table 5: A simple rate limiter.

sion challenge in verifying dynamic systems such as slicing [48, 50], symbolic execution [48, 57, 63], and abstraction refinement [53]. Our work shares similarities with these in terms of the high-level techniques for scaling verification (*e.g.,* we also use symbolic modeling and abstraction). As our targeted properties (liveness vs. safety invariants) are different, however, our application of these techniques diverges from existing works, *e.g.,* we deploy symbolic representation not to abstract packet header fields (as NoD does for verifying reachability invariants in dynamic networks [40]), but to abstract away packets altogether. Works on verifying networks via testing [24] and simulations [25, 51] are complementary to our approach. Via applying model checking, we strive to provide a fully *automatic* verifier that, unlike testing and simulation, searches the state space of our abstraction *exhaustively*.

6 Limitations and Future Work

The focus of our work is on functional correctness; this leaves out large sets of functions and properties, including those focused on path and traffic engineering properties (is a path congested? is the load balanced across multi-paths? *etc.*). Our one-big-switch abstraction is not suited for programming such functions. Plus, while a familiar abstraction to operators, the one-big-switch abstraction is relatively low-level. An interesting direction for future research is developing higher-level abstractions amenable to efficient liveness verification.

Our verifier is not a "full-stack" one; it is not designed to verify low-level properties such as memory safety and crash freedom that tools such as Vigor [62] and VigNAT [63][9] can verify and does not verify the compiled P4 code. Consequently, tools such as compiler verifiers (such as p4v [39]), P4 debuggers (*e.g.,* Vera [56])), and testers are still essential to guarantee the faithful implementation of our verified abstractions, *e.g.,* to detect and debug switch firmware and P4 compiler bugs. Finally, while our packet-less modeling improves the verification time compared to a packet-based model and enables the verification of complex properties such as liveness ones, the absolute verification times remain high for large-scale networks (note the logarithmic scale in figures). Further reducing the verification complexity of stateful functions remains an open challenge.

7 Conclusion

Modern networks rely on a variety of stateful network functions to implement rich policies. Correct operation of such networks relies on ensuring that they support key liveness properties. Unfortunately, despite exciting recent work on network verification, no existing approach is practical for, or applicable to, validating liveness. We take a top-down approach to this problem by first designing a new programming model built with verification in mind. It offers natural extensions to the convenient one-big-switch abstraction and allows decomposition of different network functions. We develop a novel encoding of these programs under dynamic events such as network state changes using Boolean formulas that can capture rich semantics (*e.g.,* counters) while also ensuring that the encoding remains bounded-size and amenable to fast liveness verification. We develop a compiler that translates our programs into those runnable on modern hardware. Our evaluation shows that the programming model can succinctly express a variety of functions, our compilation is fast, our encoding is compact and orders of magnitude more scalable to verify than naive encodings, *i.e.,* it results in substantial verification speedup.

Acknowledgments: We thank our shepherd, Meg Walraed-Sullivan, Nate Foster, Loris D'Antoni, Aws Albarghouthi, and the anonymous reviewers for their valuable comments. We gratefully acknowledge the support of the National Science Foundation through grant CNS 1910821.

[9]VigNAT [63] partitions programs into stateful and stateless components. While the stateless component is verified automatically via applying symbolic model checking, the verification of the stateful part is done via theorem proving and requires human assistance. In Vigor [62], the function code that cannot be symbolically executed is stored in a specialized library and verified by experts using theorem proving (*i.e.,* writing proofs).

References

[1] Floodlight forwarding module. https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/9142336/Forwarding.

[2] OpenFlow. openflow.org.

[3] P4 behavioral model repository. https://github.com/p4lang/behavioral-model.

[4] Project Floodlight. http://www.projectfloodlight.org/floodlight/.

[5] Al-Shabibi, A., De Leenheer, M., Gerola, M., Koshibe, A., Parulkar, G., Salvadori, E., and Snow, B. OpenVirteX: Make your virtual SDNs programmable. In *HotSDN* (2014).

[6] Alpern, B., and Schneider, F. B. Recognizing safety and liveness. *Distributed Computing 2*, 3 (1987).

[7] Arashloo, M. T., Koral, Y., Greenberg, M., Rexford, J., and Walker, D. SNAP: Stateful network-wide abstractions for packet processing. In *SIGCOMM* (2016).

[8] Baier, C., and Katoen, J.-P. *Principles of model checking*. 2008.

[9] Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., and Valadarsky, A. VeriCon: Towards verifying controller programs in software-defined networks. In *Sigplan Notices* (2014), vol. 49.

[10] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. A general approach to network configuration verification. In *SIGCOMM* (2017).

[11] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. Control plane compression. In *SIGCOMM* (2018).

[12] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *IMC* (2010).

[13] Borders, K., Springer, J., and Burnside, M. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security Symposium* (2012).

[14] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. P4: Programming protocol-independent packet processors. *SIGCOMM CCR 44*, 3 (2014), 87–95.

[15] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. Symbolic model checking: 10ˆ20 states and beyond. *Inf. Comput. 98*, 2 (1992).

[16] Carli, L. D., Sommer, R., and Jha, S. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *CCS* (2014).

[17] Chechik, M., and Paun, D. O. Events in property patterns. In *SPIN* (1999).

[18] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. NUSMV: A new symbolic model verifier. In *CAV* (1999).

[19] Clarke, E. M., Grumberg, O., and Peled, D. *Model Checking*. MIT press, 1999.

[20] Clarke, E. M., Henzinger, T. A., Veith, H., and Bloem, R. P. *Handbook of model checking*. 2016.

[21] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering* (1999).

[22] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T. D., Sekar, V., and Varghese, G. Efficient network reachability analysis using a succinct control plane representation. In *OSDI* (2016).

[23] Fayaz, S. K., Tobioka, Y., Sekar, V., and Bailey, M. Bohatei: Flexible and elastic DDoS defense. In *USENIX Security Symposium* (2015).

[24] Fayaz, S. K., Yu, T., Tobioka, Y., Chaki, S., and Sekar, V. BUZZ: Testing context-dependent policies in stateful networks. In *NSDI* (2016).

[25] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. D. A general approach to network configuration analysis. In *NSDI* (2015).

[26] Gember-Jacobson, A., Viswanathan, R., Akella, A., and Mahajan, R. Fast control plane analysis using an abstract representation. In *SIGCOMM* (2016).

[27] Ghorbani, S., and Caesar, M. Walk the line: Consistent network updates with bandwidth guarantees. In *HotSDN* (2012).

[28] Ghorbani, S., Schlesinger, C., Monaco, M., Keller, E., Caesar, M., Rexford, J., and Walker, D. Transparent, live migration of a software-defined network. In *SoCC* (2014).

[29] Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., and Wattenhofer, R. Achieving high utilization with software-driven WAN. In *SIGCOMM* (2013).

[30] Jayaraman, K., Bjørner, N., Padhye, J., Agrawal, A., Bhargava, A., Bissonnette, P.-A. C., Foster, S., Helwer, A., Kasten, M., Lee, I., et al. Validating datacenters at scale. In *SIGCOMM* (2019).

[31] John, J. P., Katz-Bassett, E., Krishnamurthy, A., Anderson, T., and Venkataramani, A. Consensus routing: The Internet as a distributed system. In *NSDI* (2008).

[32] Kang, N., Liu, Z., Rexford, J., and Walker, D. Optimizing the one big switch abstraction in software defined networks. In *CoNEXT* (2013).

[33] Kazemian, P., Chan, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[34] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[35] Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., and Clark, R. J. Kinetic: Verifiable dynamic network control. In *NSDI* (2015).

[36] Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Ingram, P., Jackson, E. J., et al. Network virtualization in multi-tenant datacenters. In *NSDI* (2014).

[37] Lamport, L. What good is temporal logic? In *IFIP congress* (1983), vol. 83.

[38] Liu, H. H., Wu, X., Zhang, M., Yuan, L., Wattenhofer, R., and Maltz, D. zUpdate: Updating data center networks with zero loss. In *SIGCOMM* (2013).

[39] Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Caşcaval, C., McKeown, N., and Foster, N. P4v: Practical verification for programmable data planes. In *SIGCOMM* (2018).

[40] Lopes, N. P., Bjørner, N., Godefroid, P., Jayaraman, K., and Varghese, G. Checking beliefs in dynamic networks. In *NSDI* (2015).

[41] Ludwig, A., Rost, M., Foucard, D., and Schmid, S. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *HotNets* (2014).

[42] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P., and King, S. T. Debugging the data plane with Anteater. In *SIGCOMM* (2011).

[43] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P., and King, S. T. VeriFlow: Verifying network-wide invariants in real time. In *NSDI* (2013).

[44] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing software defined networks. In *NSDI* (2013).

[45] Moshref, M., Bhargava, A., Gupta, A., Yu, M., and Govindan, R. Flow-level state transition as a new switch primitive for SDN. In *HotSDN* (2014).

[46] Nelson, T., Ferguson, A. D., Scheer, M. J., and Krishnamurthi, S. Tierless programming and reasoning for software-defined networks. In *NSDI* (2014).

[47] Owicki, S., and Lamport, L. Proving liveness properties of concurrent programs. *TOPLAS 4*, 3 (1982).

[48] Panda, A., Lahav, O., Argyraki, K. J., Sagiv, M., and Shenker, S. Verifying reachability in networks with mutable datapaths. In *NSDI* (2017).

[49] Pfaff, B., Pettit, J., Koponen, T., Jackson, E. J., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al. The design and implementation of Open vSwitch. In *NSDI* (2015).

[50] Plotkin, G. D., Bjørner, N., Lopes, N. P., Rybalchenko, A., and Varghese, G. Scaling network verification using symmetry and surgery. In *SIGPLAN Notices* (2016).

[51] Quoitin, B., and Uhlig, S. Modeling the routing of an autonomous system with C-BGP. *IEEE Network 19*, 6 (2005).

[52] Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. Abstractions for network update. In *SIGCOMM* (2012).

[53] Ryzhyk, L., Bjørner, N., Canini, M., Jeannin, J.-B., Schlesinger, C., Terry, D. B., and Varghese, G. Correct by construction networks using stepwise refinement. In *NSDI* (2017).

[54] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. Packet transactions: High-Level programming for line-rate switches. In *SIGCOMM* (2016).

[55] Sivaraman, A., Kim, C., Krishnamoorthy, R., Dixit, A., and Budiu, M. DC.P4: Programming the forwarding plane of a data-center switch. In *SOSR* (2015).

[56] Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L., and Raiciu, C. Debugging P4 programs with Vera. In *SIGCOMM* (2018).

[57] Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. Symnet: Scalable symbolic execution for modern networks. In *SIGCOMM* (2016).

[58] Subramanian, K., D'Antoni, L., and Akella, A. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *POPL* (2017).

[59] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., and Hudak, P. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM* (2013).

[60] Weitz, K., Woos, D., Torlak, E., Ernst, M. D., Krishnamurthy, A., and Tatlock, Z. Bagpipe: Verified BGP configuration checking. In *OOPSLA* (2016).

[61] Xie, G. G., Zhan, J., Maltz, D. A., Zhang, H., Greenberg, A., Hjalmtysson, G., and Rexford, J. On static reachability analysis of IP networks. In *INFOCOM* (2005).

[62] Zaostrovnykh, A., Pirelli, S., Iyer, R., Rizzo, M., Pedrosa, L., Argyraki, K., and Candea, G. Verifying software network functions with no verification expertise. In *SOSP* (2019).

[63] Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K., and Candea, G. A formally verified NAT. In *SIGCOMM* (2017).

8 Appendix

Table 1 lists the functions that can be represented in our language. Below we write these functions on our one-big-switch abstraction.

Simple counter: A packet counter for every source, destination IP address pair (Table 6).

Port knocking: Open a certain port *O* by attempting to open a connection on a prespecified closed port *K* (Table 7).

Simple firewall: Allows traffic between certain source, destination MAC addresses (Table 8).

IP rewrite: Rewrite IP address of all traffic coming from and directed to a particular IP address (Table 9).

Simple rate limiter: There are multiple implementations of this function. The first example (Table 5) assumes different ports are capable of sending traffic at different rates. It uses per-source IP counters to decide the traffic rate.

The second example (Table 10) uses counters on subsets of traffic to decide what traffic will be forwarded normally, redirected to a rate limiter, or dropped.

Firewall/ACL: Allows or Blocks traffic based on certain packet fields (Table 11).

Phishing/Spam detection: A per-MTA (Mail Transfer Agent) counter to detect MTAs that send a large amount of mail (Table 12).

Simple stateful firewall: A firewall that allows only the traffic whose connection was initiated by a host in *I*, where *I* is the set of departmental addresses (Table 13).

FTP monitor: Allows traffic on data port only if it received a signal on the control port (Table 14).

Heavy-hitter detection: A per-source IP counter that is incremented for every new SYN. It starts dropping packets when the counter exceeds a threshold value (Table 4).

Super spreader detection: Similar to heavy-hitter detection, the counter is incremented for every SYN. But it is also decremented for every FIN.

Sampling based on the flow size: This can be done using two tables. The first table (Table 15) uses counters to classify the flow size into three categories - small, medium, and large. It adds this metadata information into the packet (*e.g.,* using the QoS field). The second table (Table 16) samples packets based on its flow size, using its own counters.

Elephant flow detection: This is similar to sampling based on flow size, where flows of large size are elephant flows.

DNS amplification mitigation: Allows DNS reply (source port=53) to a particular IP only if it receives a DNS request/query from that IP (Table 17).

UDP flood mitigation: A per-source IP counter that is incremented for every UDP packet. It starts dropping packets when the counter exceeds a threshold value (Table 2).

Application chaining: Our language can represent non-linear chaining of applications. For example, consider a system that wants to rate-limit phishing and heavy-hitter traffic. This can be represented in our language using three tables.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & (destination IP=B) & ($0 \leq c_0 <$ m)	send()

Table 6: Simple counter.

Rule	Init	Priority	Match	Action
R_0	true	100	(source MAC=M) & (destination port=K)	add(R_1),add(R_2)
R_1	false	100	(source MAC=M) & (destination port=O)	send()
R_2	false	100	(destination MAC=M) & (destination port=O)	send()

Table 7: Port knocking.

Rule	Init	Priority	Match	Action
R_0	true	100	(source MAC=A) & (destination MAC=B)	send()
R_1	true	100	(source MAC=B) & (destination MAC=A)	send()

Table 8: Simple firewall.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A)	modify(source IP=X),send()
R_1	true	100	(destination IP=X)	modify(destination IP=A),send()

Table 9: IP rewrite.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & ($0 \leq c_0 < v_1$)	send()
R_1	true	100	(source IP=A) & ($v_1 \leq c_0 < v_2$)	send(rate limiter)
R_2	true	100	(source IP=A) & ($v_2 \leq c_0 <$ m)	drop()

Table 10: Simple rate limiter 2.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A1) & (destination IP=B1) & (source MAC=M1) & (destination MAC=N1) & (source port=P1) & (destination port=Q1)	send()
R_1	true	100	(source IP=A2) & (destination IP=B2) & (source MAC=M2) & (destination MAC=N2) & (source port=P2) & (destination port=Q2)	drop()

Table 11: Floodlight firewall/ACL.

Rule	Init	Priority	Match	Action
R_0	true	100	SMTP.MTA=A	add(R_1),add(R_2),delete(R_0),send()
R_1	false	100	(SMTP.MTA=A) & $0 \leq c_0 <$ X	send()
R_2	false	100	(SMTP.MTA=A) & X $\leq c_0 <$ m	drop()

Table 12: Phishing/spam detection.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=I) & (destination IP=E)	add(R_1),delete(R_0),send()
R_1	false	100	(destination IP=I) & (source IP=E)	send()
R_2	true	50	source IP=I	drop()

Table 13: Simple stateful firewall.

The first table does phishing detection (Table 12), the second one does heavy-hitter detection (Table 4), and third one does rate-limiting (Table 5). The first table sends suspicious traffic to the third table (instead of $drop()$) and normal traffic to the second table (instead of $send()$). The second table sends suspicious traffic to the third table (instead of $drop()$) and forwards

Rule	Init	Priority	Match	Action
R_0	true	100	(destination port=$port_{control}$) & (source IP=A) & (destination IP=B)	add(R_1),delete(R_0),send()
R_1	false	100	(source port=$port_{data}$) & (source IP=B) & (destination IP=A)	send()

Table 14: FTP monitor.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & (destination IP=B) & ($0 \leq c_0 < v_1$)	modify(flow=small),send(Sampler Table 2)
R_1	true	100	(source IP=A) & (destination IP=B) & ($v_1 \leq c_0 < v_2$)	modify(flow=medium),send(Sampler Table 2)
R_2	true	100	(source IP=A) & (destination IP=B) & ($v_2 \leq c_0 <$ m)	modify(flow=large),send(Sampler Table 2)

Table 15: Sampling based on the flow size - Sampler Table 1.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & (destination IP=B) & (flow=small) & ($c_0 = 5$)	$c_0 = 0$,send()
R_1	true	100	(source IP=A) & (destination IP=B) & (flow=medium) & ($c_0 = 500$)	$c_0 = 0$,send()
R_2	true	100	(source IP=A) & (destination IP=B) & (flow=large) & ($c_0 = 50000$)	$c_0 = 0$,send()

Table 16: Sampling based on the flow size - Sampler Table 2.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & (destination IP=B) & (destination port=53)	delete(R_0),add(R_2),add(R_3),send()
R_1	true	10	source port=53	drop()
R_2	false	100	(source IP=A) & (destination IP=B) & (destination port=53)	send()
R_3	false	100	(source IP=A) & (destination IP=B) & (source port=53)	send()

Table 17: DNS amplification mitigation.

non-suspicious traffic normally using $send()$ action.

Sol: Fast Distributed Computation Over Slow Networks

Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, Mosharaf Chowdhury
University of Michigan

Abstract

The popularity of big data and AI has led to many optimizations at different layers of distributed computation stacks. Despite – or perhaps, because of – its role as the narrow waist of such software stacks, the design of the execution engine, which is in charge of executing every single task of a job, has mostly remained unchanged. As a result, the execution engines available today are ones primarily designed for low latency and high bandwidth datacenter networks. When either or both of the network assumptions do not hold, CPUs are significantly underutilized.

In this paper, we take a first-principles approach toward developing an execution engine that can adapt to diverse network conditions. Sol, our *federated execution engine architecture*, flips the status quo in two respects. First, to mitigate the impact of high latency, Sol proactively assigns tasks, but does so judiciously to be resilient to uncertainties. Second, to improve the overall resource utilization, Sol decouples communication from computation internally instead of committing resources to both aspects of a task simultaneously. Our evaluations on EC2 show that, compared to Apache Spark in resource-constrained networks, Sol improves SQL and machine learning jobs by 16.4× and 4.2× on average.

1 Introduction

Execution engines form the narrow waist of modern data processing software stacks (Figure 1). Given a user-level intent and corresponding input for a job – be it running a SQL query on a commodity cluster [9], scientific simulations on an HPC environment [52], realtime stream processing [12], or training an AI/ML algorithm across many GPUs [7] – an execution engine orchestrates the execution of tasks across many distributed workers until the job runs to completion even in the presence of failures and stragglers.

Modern execution engines have primarily targeted datacenters with low latency and high bandwidth networks. The absence of noticeable network latency has popularized the *late-binding* task execution model in the control plane [10, 36, 43, 48] – pick the worker which will run a task only when the worker is ready to execute the task – which maximizes flexibility. At the same time, the impact of the network on task execution time is decreasing with increasing network bandwidth; most datacenter-scale applications today are compute- or memory-bound [7, 42]. The availability of high bandwidth has led to *tight coupling* of a task's roles to hide design complexity in the data plane, whereby the same task reads remote input and computes on it too. Late-binding before execution and tight coupling during execution work well together when the network is well-provisioned.

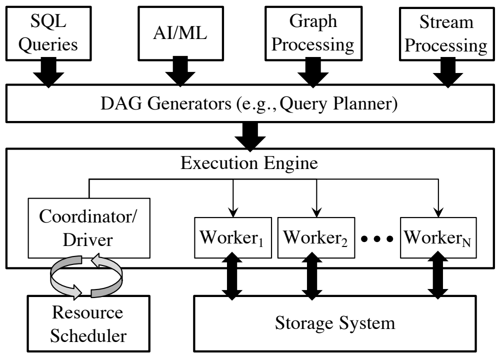


Figure 1: ***Execution engine forms the narrow waist between diverse applications and resources.***

Many emerging workloads, however, have to run on networks with high latency, low bandwidth, or both. Large organizations often perform interactive SQL and iterative machine learning between on- and off-premise storage [4, 16, 24, 27]. For example, Google uses federated model training on globally distributed data subject to privacy regulations [11, 58]; telecommunications companies perform performance analysis of radio-access networks (RAN) [30, 31]; while others troubleshoot their appliances deployed in remote client sites [39]. Although these workloads are similar to those running within a datacenter, the underlying network can be significantly constrained in bandwidth and/or latency (§2). In this paper, *we investigate the impact of low bandwidth and high latency on latency-sensitive interactive and iterative workloads.*

While recent works have proposed solutions for bandwidth-sensitive workloads, the impact of network constraints on latency-sensitive workloads has largely been overlooked. Even for bandwidth-sensitive workloads, despite many resource schedulers [28, 56], query planners [45, 50], or application-level algorithms [24, 57], the underlying execution engines of existing solutions are still primarily the ones designed for datacenters. For example, Iridium [45], Tetrium [28], and Pixida [33] rely on the execution engine of Apache Spark [54], while many others (e.g., Clarinet [50], Geode [51]) are built atop the execution engine of Apache Tez [47].

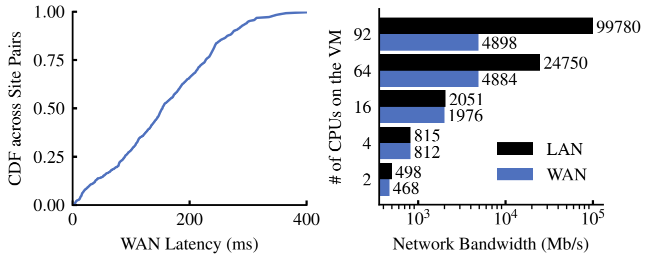
Unfortunately, under-provisioned networks can lead to large CPU underutilization in today's execution engines. First, in a high-latency network, late-binding suffers significant coordination overhead, because workers will be blocked on receiving updates from the coordinator; this leads to wasted CPU cycles and inflated completion times of latency-sensitive tasks. Indeed, late-binding of tasks to workers over the WAN can slow down the job by 8.5×–30× than running it within the local-area network (LAN). Moreover, for bandwidth-intensive tasks, coupling the provisioning of communication and computation resources at the beginning of a task's execution leads to head-of-line (HOL) blocking: bandwidth-sensitive jobs hog CPUs even though they bottleneck on data transfers, which leads to noticeable queuing delays for the rest.

By accounting for network conditions, we present a federated execution engine, Sol, which is API-compatible with Apache Spark [54]. Our design of Sol, which can transparently run existing jobs and WAN-aware optimizations in other layers of the stack, is based on two high-level insights to achieve better job performance and resource utilization.

First, we advocate *early-binding control plane decisions over the WAN* to save expensive round-trip coordinations, while continuing to late-bind workers to tasks within the LAN for the flexibility of decision making. By promoting early-binding in the control plane, we can pipeline different execution phases of the task. In task scheduling, we subscribe tasks for remote workers in advance, which creates a tradeoff: binding tasks to a remote site too early may lead to sub-optimal placement due to insufficient knowledge, but deferring new task assignments until prior tasks complete leaves workers waiting for work to do, thus underutilizing them. Our solution deliberately balances efficiency and flexibility in scheduling latency-bound tasks, while retaining high-quality scheduling for latency-insensitive tasks even under uncertainties.

Second, *decoupling the provisioning of resources for communication and computation within data plane task executions* is crucial to achieving high utilization. By introducing dedicated communication tasks for data reads, Sol decouples computation from communication and can dynamically scale down a task's CPU requirement to match its available bandwidth for bandwidth-intensive communications; the remaining CPUs can be redistributed to other jobs with pending computation.

Our evaluations show that Sol can automatically adapt to diverse network conditions while largely improving application-level job performance and cluster-level resource utilization. Using representative industry benchmarks on a 40-machine EC2 cluster across 10 regions, we show that Sol speeds up SQL and machine learning jobs by 4.9× and 16.4× on average in offline and online settings, respectively, compared to Apache Spark in resource-constrained networks. Even in datacenter environments, Sol outperforms Spark by 1.3× to 3.9×. Sol offers these benefits while effectively handling uncertainties and gracefully recovering from failures.



(a) Pairwise latency across 44 DCs. (b) Measured bandwidth on EC2.

Figure 2: ***While execution engines are widely deployed on cloud platforms, the underlying network conditions can be diverse in latency and bandwidth.***

2 Background and Motivation

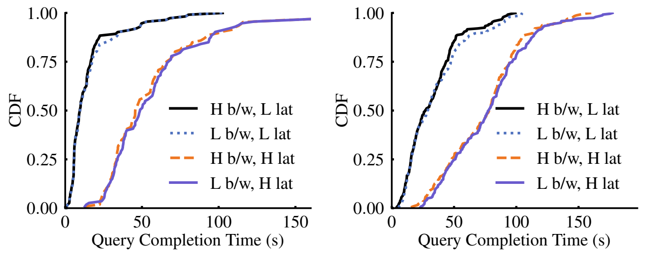
2.1 Execution Engines

The execution engine takes a graph of *tasks* – often a directed acyclic graph (DAG) – from the higher-level scheduler as its primary input. Tasks performing the same computation function on different data are often organized into *stages*, with dependencies between the stages represented by the edges of the execution DAG. Typically, a central *coordinator* in the execution engine – often referred to as the driver program of a job – interacts with the cluster resource manager to receive required resources and spawns *workers* across one or more machines to execute runnable tasks.[1] As workers complete tasks, they notify the coordinator to receive new runnable tasks to execute.

Design space. The design of an execution engine should be guided by how the environment and workload characteristics affect delays in the *control plane* (i.e., coordinations between the coordinator and workers as well as amongst the workers) and in the *data plane* (i.e., processing of data by workers). Specifically, a task's lifespan consists of four key components:

- *Coordination time* (t_{coord}) represents the time for orchestrating task executions across workers. This is affected by two factors: network latency, which can vary widely between different pairs of sites (Figure 2(a)), and the inherent computation overhead in making decisions. While the latter can be reduced by techniques like reusing schedules [37,49], the former is determined by the environment.
- *Communication time* (t_{comm}) represents the time spent in reading input and writing output of a task over the network and to the local storage.[2] For the same amount of data, time spent in communication can also vary widely based on Virtual Machine (VM) instance types and LAN-vs-WAN (Figure 2(b)).
- *Computation time* (t_{comp}) represents the time spent in running every task's computation.
- *Queuing time* (t_{queue}) represents the time spent waiting

[1]A task becomes runnable whenever its dependencies have been met.
[2]Most transfers after the input-reading stages happen over the network.



(a) Apache Spark (b) Apache Tez

Figure 3: ***TPC query completion times in different network settings using different execution engines (scale factor is set to 100).***

for resource availability before execution. Given a fixed amount of resources, tasks of one job may have to wait for tasks of other jobs to complete.

We first take into account t_{comp}, t_{coord}, and t_{comm} in characterizing the design of execution engines for a single task. By assuming $t_{comp} \gg t_{coord}, t_{comm}$ (i.e., by focusing on the execution of *compute-bound* workloads such as HPC [21], AI training [7] and in many cases within datacenters), existing execution engines have largely ignored two settings in the design space.

First, the performance of jobs can be dominated by the coordination time (i.e., $t_{coord} \gg t_{comm}, t_{comp}$), and more time is spent in the control plane than the data plane. An example of such a scenario within a datacenter would be stream processing using mini-batches, where scheduling overhead in the coordinator is the bottleneck [49]. As $t_{coord} \to O(100)$ ms over the WAN, coordination starts to play a bigger role even when scheduler throughput is not an issue. As $\frac{t_{comp}}{t_{coord}}$ and $\frac{t_{comm}}{t_{coord}}$ decrease, e.g., in interactive analytics [30, 31] and federated learning [11], coordination time starts to dominate the end-to-end completion time of each task.

Second, in *bandwidth-bound* workloads, more time is likely to be spent in communication than computation (i.e., $t_{comm} > t_{coord}, t_{comp}$). Examples of such a scenario include big data jobs in resource-constrained private clusters [36] or across globally distributed clouds [24, 45, 50, 51], and data/video analytics in a smart city [25].

In the presence of multiple jobs, inefficiency in one job's execution engine can lead to inflated t_{queue} for other jobs' tasks. For latency-sensitive jobs waiting behind bandwidth-sensitive ones, t_{queue} can quickly become non-negligible.

2.2 Inefficiencies in Constrained Network Conditions

While there is a large body of work reasoning about the performance of existing engines in high-bandwidth, low-latency datacenters [41, 42], the rest of the design space remains unexplored. We show that existing execution engines suffer significant resource underutilization and performance loss in other settings.

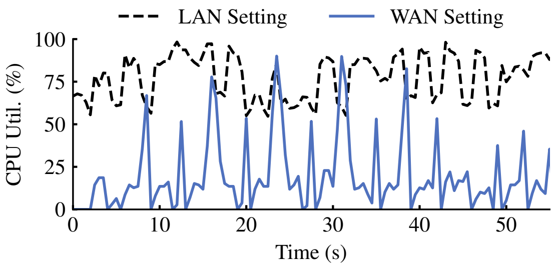


Figure 4: ***CPU utilization throughout a machine learning job.***

Performance degradation due to high latency. To quantify the impact of high latency on job performance, we analyzed the individual query completion times of 110 queries on two industrial benchmarks: TPC-DS and TPC-H. We use two popular execution engines – Apache Spark [54] and Apache Tez [47] – on a 10-site deployment; each site has 4 machines, each with 16 CPU cores and 64 GB of memory. We consider four network settings, each differing from the rest in terms of either bandwidth or latency as follows: [3]

- *Bandwidth:* Each VM has a 10 Gbps NIC in the high-bandwidth and 1 Gbps in the low-bandwidth setting.
- *Latency:* Latency across machines is <1 ms in the low-latency setting, while latencies across sites vary from $O(10)$–400 ms in the high-latency setting.

Figure 3 shows the distributions of average query completion times of Spark and Tez, where we use a dataset of scale factor 100.[4] We found that the availability of more bandwidth has little impact on query completion times; different query plans and task placement decisions in Spark and Tez did not improve the situation either. However, job completion times in the high-latency setting are significantly inflated – up to 20.6× – than those in the low-latency setting. Moreover, we observe high network latency can lead to inefficient use of CPUs for a latency-bound machine learning job (Figure 4).

The root cause behind this phenomenon is the *late-binding* of tasks to workers in the control plane. In existing execution engines, decision making in the control plane, such as task scheduling [50] and straggler mitigation [8], often requires realtime information from data plane executions, whereas data processing is initiated by control plane decisions. With high coordination latency, this leads to wasted CPU cycles as each blocks on acquiring updates from the other.

CPU underutilization due to low bandwidth. To understand the impact of low bandwidth on resource efficiency, we analyzed bandwidth-sensitive workloads using a scale factor of 1000 on the same experimental settings as above.

Figure 5 reports both the CPU and network utilizations throughout the execution of a representative query (query-

[3]We use the latency profile of 10 sites on EC2 and set a large TCP window size to reach the network capacity [34]. For the high-bandwidth setting and the low-bandwidth one, we refer to the available LAN bandwidth on *m4.10xlarge* and *m4.2xlarge* instances, respectively.

[4]A scale factor of X means a X GB dataset.

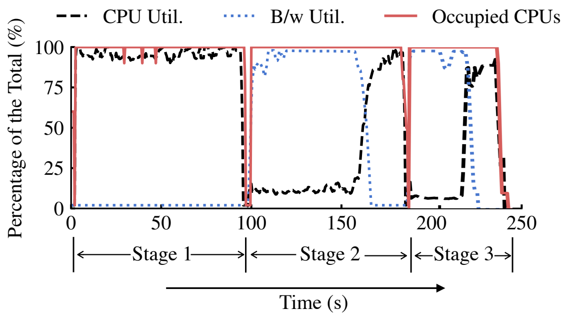


Figure 5: ***Resource utilization over a bandwidth-bound query's lifespan (scale factor is set to 1000).***

25 from the TPC-DS benchmark), which involves two large shuffles (in stage 2 and stage 3) over the network. During task executions, large data reads over the network are communication-intensive, while computations on the fetched data are CPU-intensive. We observe that, when tasks are bandwidth-constrained, their overall CPU utilization plummets even though they continue to take up all the available CPUs. This is due to the coupling of communication with computation in tasks. In other words, the number of CPUs involved in communication is independent of the available bandwidth. The end result is head-of-line (HOL) blocking of both latency- and bandwidth-sensitive jobs (not shown) by bandwidth-bound underutilized CPUs of large jobs.

Shortcomings of existing works. Existing works on WAN-aware query planning and task placement [28, 45, 50, 51] cannot address the aforementioned issues because they focus on managing and/or minimizing bandwidth usage during task execution, not on the impact of latency before execution starts or CPU usage during task execution.

3 Sol: A Federated Execution Engine

To address the aforementioned limitations, we present Sol, a *federated execution engine* which is aware of the underlying network's characteristics (Figure 6). It is primarily designed to facilitate efficient execution of emerging distributed workloads across a set of machines which span multiple sites (thus, have high latency between them) and/or are interconnected by a low bandwidth network. Sol assumes that machines within the same site are connected over a low-latency network. As such, it can perform comparably to existing execution engines when deployed within a datacenter.

Design goals. In designing Sol, we target a solution with the following properties:

- *High-latency coordinations should be pipelined.* Coordinations between control and data planes should not stall task executions. As such, Sol should avoid synchronous coordination (e.g., workers blocking to receive tasks) to reduce overall t_{coord} for latency-bound tasks. This leads to early-binding of tasks over high-latency networks.
- *Underutilized resources should be released.* Sol should release unused resources to the scheduler, which can be repurposed to optimize t_{queue} for pending tasks. This calls for decoupling communication from computation in the execution of bandwidth-intensive tasks without inflating their t_{comm} and t_{comp}.
- *Sol should adapt to diverse environments automatically.* The network conditions for distributed computation can vary at different points of the design space. Sol should, therefore, adapt to different deployment scenarios with built-in runtime controls to avoid reinventing the design.

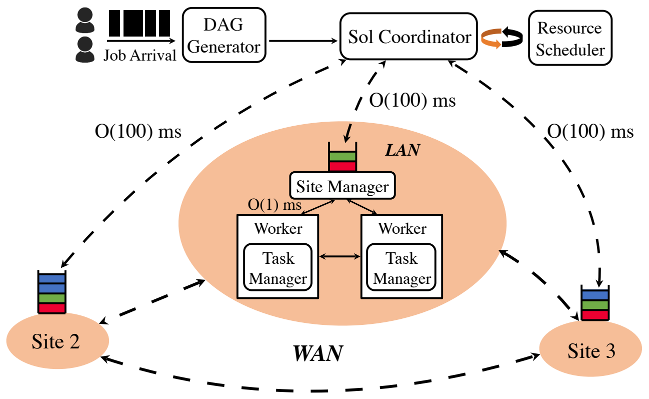


Figure 6: ***Sol components and their interactions. Low-latency sites synchronously coordinate within themselves and asynchronously coordinates across high-latency links.***

System components. At its core, Sol has three primary components:

- *Central Coordinator:* Sol consists of a logically centralized coordinator that orchestrates the input job's execution across many remote compute sites. It can be located at any of the sites; each application has its own coordinator or driver program. Similar to existing coordinators, it interacts with a resource manager for resource allocations.
- *Site Manager:* Site managers in Sol coordinate local workers within the same site. Each site manager has a shared queue, where it enqueues tasks assigned by the central coordinator to the workers in this site. This allows for late-binding of tasks to workers within the site and ensures high resource utilization, wherein decision making can inherit existing designs for intra-datacenter systems. The site manager also detects and tackles failures and stragglers that are contained within the site.
- *Task Manager:* At a high level, the task manager is the same as today: it resides at individual workers and manages tasks. However, it manages compute and communication resources independently.

Figure 7 shows a high-level overview explaining the interaction among these components throughout our design in the control plane (§4) and the data plane (§5).

4 Sol Control Plane

Modern execution engines primarily target datacenters with low latency networks [7, 12, 47, 54], wherein late-binding of

```
    ▷ Operations in Central Coordinator
 1: for Site $s$ in all sites do
 2:   while currentTaskNum($s$) < targetQueLen($s$) do    ▷ §4.3
 3:     if Exist available tasks $t$ for scheduling to $s$ then
 4:       Push $t$ to Site Manager in $s$
 5:     else
 6:       Breakdown task dependency judiciously            ▷ §4.4

    ▷ Operations in Site Manager
 7: if Receive task assignment then
 8:   Queue up task
 9: else if Receive task completion then
10:   Notify coordinator and schedule next task $t$
11:   if Task $t$ requires large remote read then
12:     Issue fetch request to the scheduled worker        ▷ §5.1
13:   else
14:     Launch task $t$
15: else if Input is ready for computation task $t$ then
16:   Activate and launch task $t$                         ▷ §5.3

    ▷ Operations in Task Manager
17: if Receive task assignment $t$ then
18:   Execute task $t$
19: else if Detect task completion then
20:   Notify Site Manager for new task assignment
21: else if Receive data fetch request then
22:   Initiate communication task                          ▷ §5.2
```

Figure 7: ***The interaction between the central coordinator, site manager and task manager.***

tasks to workers maximizes flexibility. For example, the coordinator assigns new tasks to a worker after it is notified of new resource availability (e.g., due to task completion) from that worker. Moreover, a variety of on-demand communication primitives, such as variable broadcasts and data shuffles, are also initiated lazily by the coordinator and workers. In the presence of high latency, however, late-binding results in expensive coordination overhead (§2.2).

In this section, we describe how Sol pushes tasks to sites to hide the expensive coordination latency (§4.1), the potential benefits of push-based execution (§4.2) as well as how we address the challenges in making it practical; i.e., how to determine the right number of tasks to push to each site (§4.3), how to handle dependencies between tasks (§4.4), and how to perform well under failures and uncertainties (§4.5).

4.1 Early-Binding to Avoid High-Latency Coordination

Our core idea to hide coordination latency (t_{coord}) over high-latency links is early-binding tasks to sites. Specifically, Sol optimizes t_{coord} between the central coordinator and remote workers (i) by *pushing* and queuing up tasks in each site; and (ii) by *pipelining* task scheduling and execution across tasks.

Figure 8 compares the control flow in traditional designs

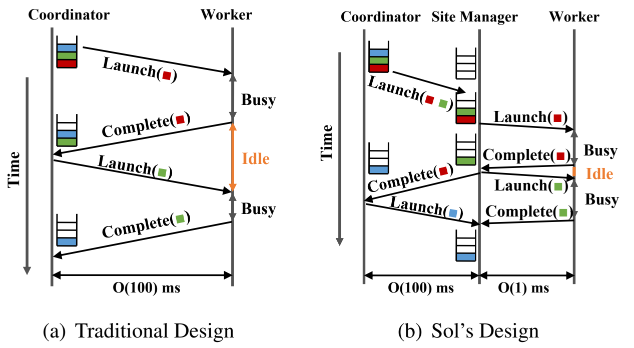


Figure 8: ***Task execution control flows in traditional designs vs. Sol. In Sol, tasks (denoted by colored rectangles) are queued at a site manager co-located with workers.***

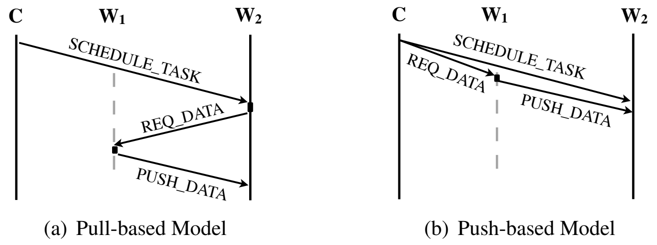


Figure 9: ***Sol adopts the push-based model to pipeline task scheduling and data fetch.***

with that in Sol. In case of late-binding, workers have to wait for new task assignments from the remote coordinator. In contrast, the site manager in Sol directly dispatches a task already queued up in its local queue and asynchronously notifies the coordinator of task completions as well as the resource status of the site. The coordinator makes new task placement decisions and queues up tasks in site managers asynchronously, while workers in that site are occupied with task executions.

Furthermore, this execution model enables us to pipeline t_{coord} and t_{comm} for each individual task's execution. When the coordinator assigns a task to a site, it notifies the corresponding upstream tasks (i.e., tasks in the previous stage) of this assignment. As such, when upstream tasks complete, they can proactively push their latency-bound output partitions directly to the site where their downstream tasks will execute, even though the control messages containing task placements may still be on-the-fly. As shown in Figure 9, pull-based data fetches experience three sequential phases of communication; in contrast, the scheduling of downstream tasks and their remote data reads are pipelined in the push-based model, improving their completion times.

4.2 Why Does Early-Binding Help?

Assume that the coordinator continuously allocates tasks to a remote site's k CPUs. For each task completion, the pull-based model takes one RTT for the coordinator to receive the task completion notification and then send the next task assignment; during this period, the task is queued up in the coordinator for scheduling. Hence, on average, $\frac{i-1}{k}$ RTTs are

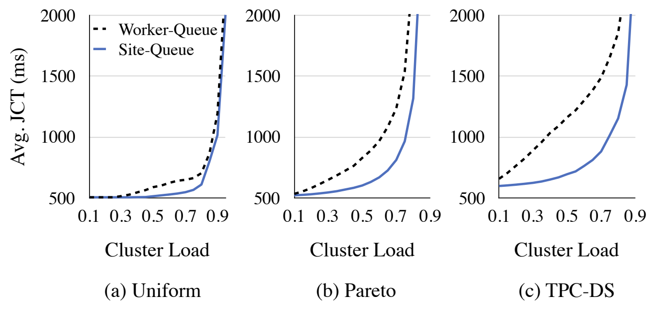


Figure 10: ***Job performance with Site- and Worker-Queue approaches. Variance in task durations increases from (a) to (c).***

wasted before the i^{th} task runs. The push-based model can save up to $\frac{i-1}{k}$ RTTs for the i^{th} task by pipelining inter-task assignments and executions. Our analysis over 44 datacenters using the measured inter-site latencies shows that, compared to late-binding, the push-based model can achieve an average improvement of 153 ms for the data fetch of every downstream task (more details in Appendix A). Such gaps become magnified at scale with a large number of small tasks, e.g., a large fan-out, latency-sensitive job.

One may consider pulling multiple tasks for individual workers at a time, but the push model provides more flexibility. Pushing from the coordinator can react to online scheduling better. When tasks arrive in an online manner, multiple tasks may not be available for pulling at a time. e.g., when a new task keeps arriving right after serving a pull request, pulling multiple tasks degenerates into pulling one by one.

Moreover, by late-binding task assignments within the site, our site-manager approach enables more flexibility than pushing tasks to individual workers (i.e., maintaining one queue per worker). To evaluate this, we ran three workloads across 10 EC2 sites, where all workloads have the same average task duration from TPC-DS benchmarks but differ in their distributions of task durations. Figure 10 shows that the site-manager approach achieves superior job performance owing to better work balance, particularly when task durations are skewed.

4.3 How to Push the Right Number of Tasks?

Determining the number of queued-up tasks for site managers is crucial for balancing worker utilization versus job completion times. On the one hand, queuing up too few tasks leads to underutilization, inflating t_{queue} due to lower system throughput. On the other hand, queuing up too many leads to sub-optimal work assignments because of insufficient knowledge when early-binding, which inflates job completion times as well (see Appendix B for more details).

Our target: Intuitively, as long as a worker is not waiting to receive work, queuing more tasks does not provide additional benefit for improving utilization. To fully utilize the resource, we expect the total execution time of the queued-up tasks will occupy the CPU before the next task assignment arrives, which is the key to strike the balance between utilization and job performance.

Our solution: When every task's duration is known, the number of queued-up tasks can adapt to the instantaneous load such that the total required execution time of the queued-up tasks keeps all the workers in a site busy, but not pushing any more to retain maximum flexibility for scheduling across sites. However, individual task durations are often highly skewed in practice [8], while the overall distribution of task durations is often stable over a short period [20, 44].

Even without presuming task-specific characteristics or distributions, we can still approximate the ideal queue length at every site dynamically for a given resource utilization target. We model the total available cycles in each scheduling round as our target, and the duration of each queued-up task is a random variable. This can be mapped into a packing problem, where we have to figure out how many random variables to sum up to achieve the targeted sum.

When the individual task duration is not available, we extend Hoeffding's inequality, and inject the utilization target into our model to determine the desired queue length (Appendix C for a formal result and performance analysis). Hoeffding's inequality is known to characterize how the sum of random variables deviates from its expected value with the minimum, the average, and the maximum of variables [23]. We extend it but filter out the outliers in tasks, wherein we rely on three statistics – 5th percentile, average, and 95th percentile (which are often stable) – of the task duration by monitoring the tasks over a period. As the execution proceeds, the coordinator in Sol inquires the model to generate the target queue size, whereby it dynamically pushes tasks to each site to satisfy the specified utilization. Note that when the network latency becomes negligible, our model outputs zero queue length as one would expect.

4.4 How to Push Tasks with Dependencies?

In the presence of task dependencies, where tasks may depend on those in their parent stage(s), pushing tasks is challenging, since it creates a tradeoff between the efficiency and quality of pipelining. For latency-sensitive tasks, we may want to push downstream tasks to save round-trip coordinations even before the upstream output is available. However, for bandwidth-intensive tasks, pushing their downstream tasks will not bring many benefits; this may even miss optimal task placements due to insufficient knowledge about the outputs from all upstream tasks [45, 50]. Sol, therefore, has to reconcile between latency- and bandwidth-sensitive tasks at runtime without presuming task properties.

To achieve desired pipelining efficiency for latency-bound tasks, Sol speculates the best placements for downstream tasks. Our straw-man heuristic is first pushing the downstream task to the site with the least work, with an aim to minimize the queueing time on the site. Moreover, Sol can refine its speculation by learning from historical trends (e.g., recurring jobs) or the iterative nature of many jobs. For example, in stream processing and machine learning, the output partitions

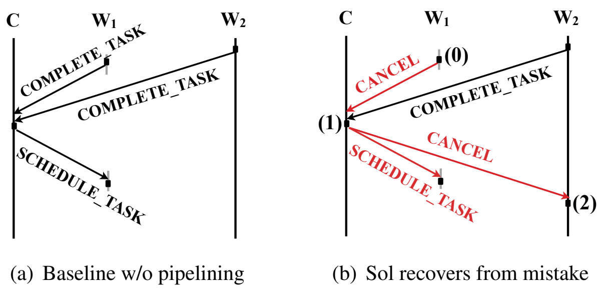


Figure 12: ***(b) The recovery process is shown in red. (0) W_1 detects large output, and sends CANCEL message to the coordinator C for task rescheduling. (1) Upon receiving the update, C waits until it gathers required information, then reschedules task, and (3) cancels task in W_2.***

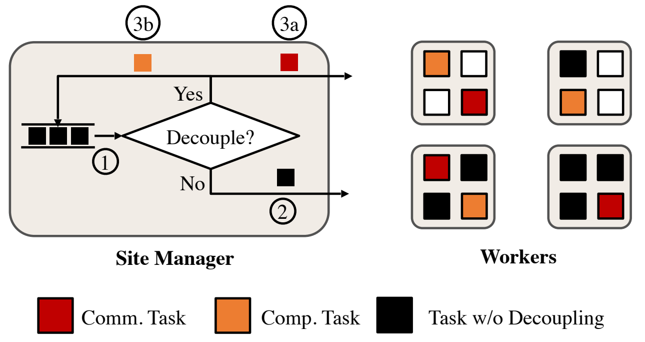


Figure 13: ***High-level overview of data plane decoupling.***

computed for every batch are largely similar [55], so are their task placements [49]. As such, Sol can reuse the placement decisions in the past run.

However, even when a bandwidth-intensive task is pushed to a suboptimal site, Sol can gracefully retain the scheduling quality via worker-initiated re-scheduling. Figure 12 shows the control flow of the recovery process in Sol and the baseline. In Figure 12(b), we push upstream tasks to workers W_1 and W_2, and downstream tasks are pushed only to W_2. As the upstream task in W_1 proceeds, the task manager detects large output is being generated, which indicates we are in the regime of bandwidth-intensive tasks. (0) Then W_1 notifies the coordinator C of task completion and a CANCEL message to initiate rescheduling for the downstream task. (1) Upon receiving the CANCEL message, the coordinator will wait until it collects output metadata from W_1 and W_2. The coordinator then reschedules the downstream task, and (2) notifies W_2 to cancel the pending downstream task scheduled previously. Note that the computation of a downstream task will not be activated unless it has gathered the upstream output.

As such, even when tasks are misclassified, Sol performs no worse than the baseline (Figure 12(a)). First, the recovery process does not introduce more round-trip coordinations due to rescheduling, so it does not waste time. Moreover, even in the worst case, where all upstream tasks have preemptively pushed data to the downstream task by mistake, the total amount of data transfers is capped by the input size of the downstream. However, note that the output is pushed only if it is latency-sensitive, so the amount of wasted bandwidth is also negligible as the amount of data transfers is latency-bound.

4.5 How to Handle Failures and Uncertainties?

Fault tolerance and straggler mitigation are enforced by the local site manager and the global coordinator. Site managers in Sol try to restart a failed task on other local workers; failures of long tasks or persistent failures of short tasks are handled via coordination with the remote coordinator. Similarly, site managers track the progress of running tasks and selectively duplicate small tasks when their execution lags behind. Sol can gracefully tolerate site manager failures by redirecting workers' control messages to the central coordinator, while a secondary site manager takes over (§7.5).

Moreover, Sol can guarantee a bounded performance loss due to early-binding even under uncertainties. To ensure a task at the site manager will not experience arbitrarily large queueing delay, the site manager can withdraw the task assignment when the task queueing delay on this site exceeds Δ. As such, the total performance loss due to early-binding is (Δ + RTT), since it takes one RTT to push and reclaim the task.

5 Decoupling in the Data Plane

In existing execution engines, the amount of CPU allocated to a task when it is reading data is the same as when it later processes the data. This tight coupling between resources leads to resource underutilization (§2.2). In this section, we introduce how to improve t_{queue} by mitigating HOL blocking.

5.1 How to Decouple the Provisioning of Resource?

To remove the coupling in resource provisioning, Sol introduces dedicated communication tasks,[5] which fetch task input from remote worker(s) and deserialize the fetched data, and computation tasks, which perform the computation on data. The primary goal of decoupling is to scale down the CPU requirements when multiple tasks have to fetch data over low-bandwidth links.

Communication and computation tasks are internally managed by Sol without user intervention. As shown in Figure 13, ① when a task is scheduled for execution, the site manager checks its required input for execution and reserves the provisioned resource on the scheduled worker. ② Bandwidth-insensitive tasks will be dispatched directly to the worker to execute. (3a) However, for tasks that need large volumes of remote data, the site manager will notify the task manager on the scheduled worker to set up communication tasks for data preparation. (3b) At the same time, the corresponding computation tasks be marked as inactive and do not start their execution right away. Once input data is ready for computation, the site manager will activate corresponding computation tasks to perform computation on the fetched data.

[5]Each communication task takes one CPU core by default in our design.

Although decoupling the provisioning of computation and communication resource will not speed up individual tasks, it can greatly improve overall resource utilization. When the input for a task's computation is being fetched by the communication task, by oversubscribing multiple computation tasks' communication to fewer communication tasks, Sol can release unused CPUs and repurpose them for other tasks. In practice, even the decoupled job can benefit from its own decoupling; e.g., when tasks in different stages can run in parallel, which is often true for jobs with complicated DAGs, computation tasks can take up the released CPUs from other stages in decoupling.

5.2 How Many Communication Tasks to Create?

Although decoupling is beneficial, we must avoid hurting the performance of decoupled jobs while freeing up CPUs. A key challenge in doing so is to create the right number of communication tasks to fully utilize the available bandwidth. Creating too many communication tasks will hog CPUs, while creating too few will slow down the decoupled job.

We use a simple model to characterize the number of required communication tasks. There are two major operations that communication tasks account for: (i) fetch data with CPU cost $C_{I/O}$ every time unit; (ii) deserialize the fetched data simultaneously with CPU cost C_{deser} in unit time. When the decoupling proceeds with I/O bandwidth B, the total requirement of communication tasks N can be determined based on the available bandwidth ($N = B \times (C_{I/O} + C_{deser})$).

Referring to the network throughput control, we use an adaptive tuning algorithm. When a new task is scheduled for decoupling, the task manager first tries to hold the provisioned CPUs to avoid resource shortage in creating communication tasks. However, the task manager will opportunistically cancel the launched communication task after its current fetch request completes, and reclaim its CPUs if launching more communication tasks does not improve bandwidth utilization any more.[6] During data transfers, the task manager monitors the available bandwidth using an exponentially weighted moving average (EWMA).[7] As such, the task manager can determine the number of communication tasks required currently: $N_{current} = \lceil \frac{B_{current}}{B_{old}} \times N_{old} \rceil$. Therefore, it will launch more communication tasks when more bandwidth is available and the opposite when bandwidth decreases. Note that the number of communication tasks is limited by the total provisioned CPUs for that job to avoid performance interference.

5.3 How to Recover CPUs for Computation?

Sol must also ensure that the end-to-end completion time on computation experiences negligible inflation. This is because when the fetched data is ready for computation, the decoupled job may starve if continuously arriving computation tasks take up its released computation resources.

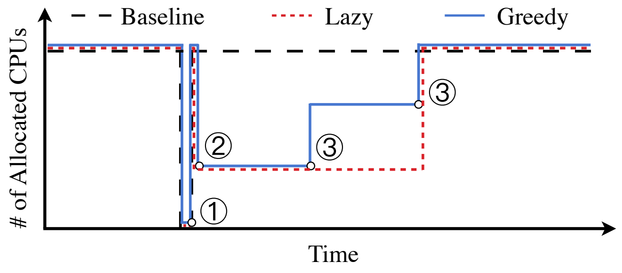


Figure 14: ***Three strategies to manage decoupled jobs. We adopt the Greedy strategy: ① when the downstream tasks start, they hold the reserved CPUs. ② But the task manager will reclaim the unused CPUs, and ③ activate the computation task once its input data is ready. The first trough marks the stage boundary.***

We refer to not decoupling as the *baseline* strategy, while waiting for the entire communication stage to finish as the *lazy* strategy. The former wastes resources, while the latter can hurt the decoupled job. Figure 14 depicts both.

Instead, Sol uses a *greedy* strategy, whereby as soon as some input data becomes ready (from upstream tasks), the site manager will prioritize the computation task corresponding to that data over other jobs and schedule it. As such, we can gradually increase its CPU allocation instead of trying to acquire all at once or holding onto all of them throughout.

5.4 Who Gets the Freed up CPUs?

Freed up CPUs from the decoupled jobs introduce an additional degree of freedom in scheduling. Resource schedulers can assign them in a FIFO or fair manner. As the duration of communication tasks can be estimated by the remaining data fetches and the available bandwidth, the scheduler can plan for the extra resources into the future, e.g., similar to [19].

6 Implementation

While our design criteria are not married to specific execution engines, we have implemented Sol in a way that keeps it API compatible with Apache Spark [2] in order to preserve existing contributions in the big data stack.

Control and Data Plane To implement our federated architecture, we add site manager modules to Spark, wherein each site manager keeps a state store for necessary metadata in task executions, and the metadata is shared across tasks to avoid redundant requests to the remote coordinator. The central coordinator coordinates with the site manager by heartbeat as well as the piggyback information in task updates. During executions, the coordinator monitors the network latency using EWMA in a one second period. This ensures that we are stable despite transient latency spikes. When the coordinator schedules a task to the site, it assigns a dummy worker for the pipelining of dependent tasks (e.g., latency-bound output will be pushed to the dummy worker). Similar to delay scheduling, we set the queueing delay bound Δ to 3 seconds [56]. Upon receiving the completion of upstream tasks, the site manager

[6] This introduces little overhead, since the data fetch is in a streaming manner, wherein the individual block is small.

[7] $B_{current} = \alpha\, B_{measured} + (1-\alpha)\, B_{old}$, where α is the smoothing factor ($\alpha = 0.2$ by default) and B denotes the available bandwidth over a period.

can schedule the downstream task more intelligently with late-binding. Meanwhile, the output information from upstream tasks is backed up in the state store until their completions.

Support for Extensions Our modifications are to the core of Apache Spark, so users can enjoy existing Spark-based frameworks on Sol without migrations of their codebase. Moreover, for recent efforts on WAN-aware optimizations, Sol can support those more educated resource schedulers or location-conscious job schedulers by replacing the default, but further performance analysis of higher-layer optimizations is out of the scope of this paper. To the best of our knowledge, Sol is the first execution engine that can optimize the execution layer across the design space.

7 Evaluation

In this section, we empirically evaluate Sol through a series of experiments using micro and industrial benchmarks. Our key results are as follows:

- Sol improves performance of individual SQL and machine learning jobs by 4.9×–11.5× w.r.t. Spark and Tez execution engines in WAN settings. It also improves streaming throughput by 1.35×–3.68× w.r.t. Drizzle (§7.2).
- In online experiments, Sol improves the average job performance by 16.4× while achieving 1.8× higher utilization (§7.3).
- Even in high bandwidth-low latency (LAN) setting, Sol improves the average job performance by 1.3× w.r.t. Spark; its improvement in low bandwidth-low latency setting is 3.9× (§7.4).
- Sol can recover from failures faster than its counterparts, while effectively handling uncertainties (§7.5).

7.1 Methodology

Deployment Setup We first deploy Sol in EC2 to evaluate individual job performance using instances distributed over 10 regions.[8] Our cluster allocates 4 *m4.4xlarge* instances in each region. Each has 16 vCPUs and 64GB of memory. To investigate Sol performance on multiple jobs in diverse network settings, we set up a 40-node cluster following our EC2 setting, and use *Linux Traffic Control* to perform network traffic shaping to match our collected profile from 10 EC2 regions.

Workloads We use three types of workloads in evaluations:

1. *SQL*: we evaluate 110 industry queries in TPC-DS/TPC-H benchmarks [5, 6]. Performance on them is a good demonstration of how good Sol would perform in real-world applications handling jobs with complicated DAGs.
2. *Machine learning*: we train three popular federated learning applications: linear regression, logistic regression, and k-means, from Intel's industry benchmark [26]. Each training data consists of 10M samples, and the training time of each iteration is dominated by computation.
3. *Stream processing*: we evaluate the maximum throughput that an execution design can sustain for *WordCount* and *TopKCount* while keeping the end-to-end latency by a given target. We define the end-to-end latency as the time from when records are sent to the system to when results incorporating them appear.

Baselines We compare Sol to the following baselines:

1. *Apache Spark* [54] and *Apache Tez* [47]: the mainstream execution engines for generic workloads in datacenter and wide-area environments.
2. *Drizzle* [49]: a recent engine tailored for streaming applications, optimizing the scheduling overhead.

Metrics Our primary metrics to quantify performance are the overarching user-centric and operator-centric objectives, including *job completion time (JCT)* and *resource utilization*.

7.2 Performance Across Diverse Workloads in EC2

In this section, we evaluate Sol's performance on individual jobs in EC2, with query processing, machine learning, and streaming benchmarks.

Sol outperforms existing engines Figure 15 shows the distribution of query completion times of 110 TPC queries individually on (10, 100, 1000) scale factor datasets. As expected, Sol and Spark outperform Tez by leveraging their in-memory executions. Meanwhile, Sol speeds up individual queries by 4.9× (11.5×) on average and 8.6× (23.3×) at the 95th percentile over Spark (Tez) for the dataset with scale factor 10. While these queries become more bandwidth- and computation- intensive as we scale up the dataset, Sol can still offer a noticeable improvement of 3.4× and 1.97× on average compared to Spark on datasets with scale factors 100 and 1000, respectively. More importantly, Sol outperforms the baselines across all queries.

Sol also benefits machine learning and stream processing. For such predictable workloads, Sol pipelines the scheduling and data communication further down their task dependencies. Figure 16 reports the average duration across 100 iterations in machine learning benchmarks, where Sol improves the performance by 2.64×-3.01× w.r.t. Spark.

Moreover, Sol outperforms Drizzle [49] in streaming workloads. Figure 17 shows that Sol achieves 1.35×–3.68× higher throughput than Drizzle. This is because Sol follows a *push-based* model in both control plane coordinations and data plane communications to pipeline round-trips for inter-site coordinations, while Drizzle optimizes the coordination overhead between the coordinator and workers. Allowing a larger target latency improves the throughout, because the fraction of computation time throughout the task lifespan increases, and thus the benefits from Sol become less relevant.

Sol is close to the upper bound performance To explore how far Sol is from the optimal, we compare Sol's perfor-

[8] California, Sydney, Oregon, Ohio, Tokyo, Mumbai, Seoul, Singapore, Sao Paulo and Frankfurt.

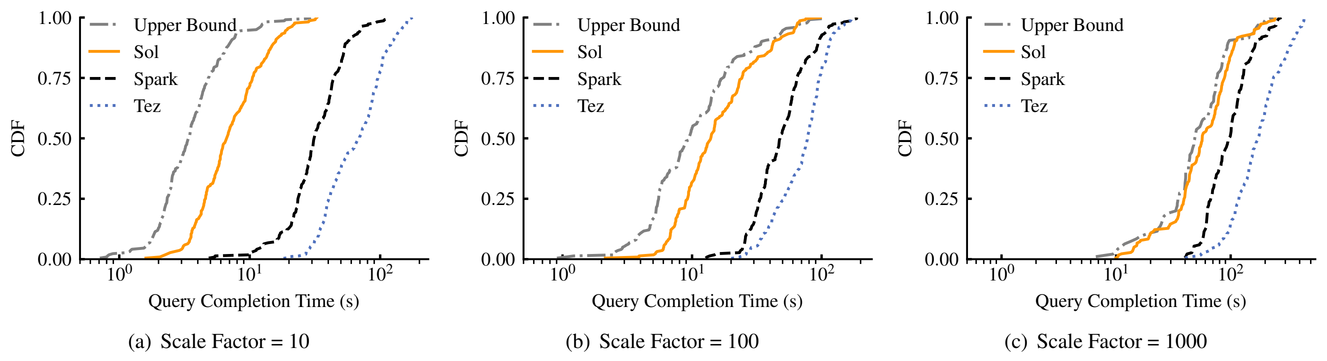


Figure 15: ***Performance of Sol, Spark, and Tez on TPC query processing benchmark.***

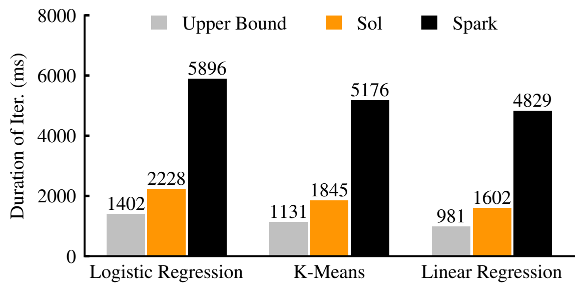


Figure 16: ***Performance on machine learning.***

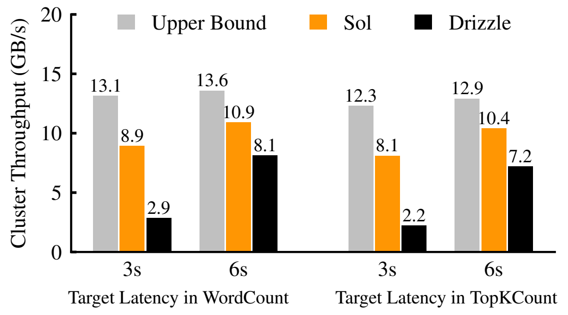


Figure 17: ***Performance on stream processing. Higher is better.***

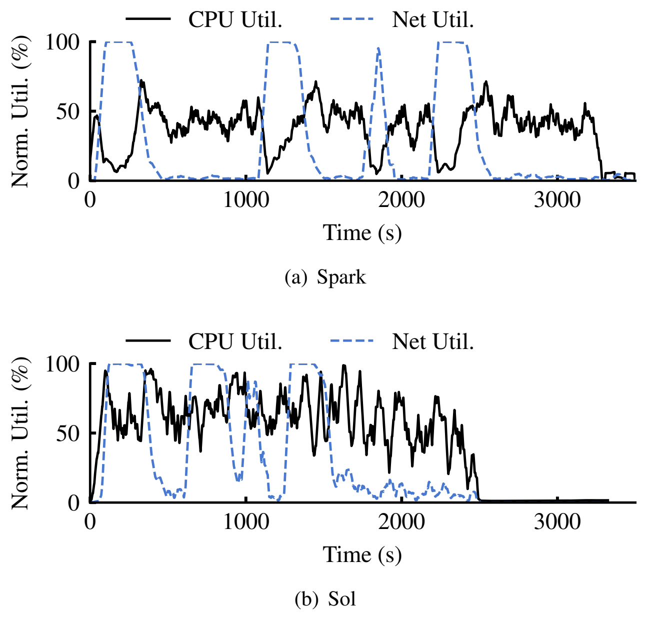


Figure 18: ***Resource utilization over time.***

mance in the high latency setting against Sol in a hypothetical latency-free setting [9], which is a straightforward upper bound on its performance. While high latencies lead to an order-of-magnitude performance degradation on Spark, Sol is effectively approaching the optimal. As shown in Figure 15 and Figure 16, Sol's performance is within 3.5× away from the upper bound. As expected in Figure 15(c), this performance gap narrows down as Sol has enough work to queue-up for hiding the coordination delay.

7.3 Online Performance Breakdown

So far we have evaluated Sol in the offline setting with individual jobs. Here we move on to evaluate Sol with diverse workloads running concurrently and arriving in an online fashion with our cluster. Specifically, we evaluate Sol in an online setting, where we run 160 TPC queries – randomly drawn from the (10, 100)-scale TPC benchmarks – run as foreground, interactive jobs, and bandwidth-intensive CloudSort jobs [3] – each has 200 GB or 1 TB GB input – in the background. The TPC queries are submitted following a Poisson process with an average inter-arrival time of 10 seconds, while the CloudSort jobs are submitted every 300 seconds.

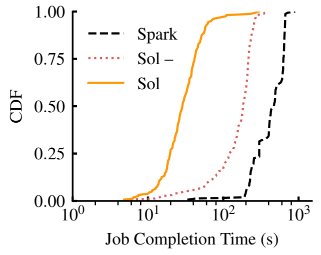
We evaluate Sol and Spark using two job schedulers:

1. *FIFO*: Jobs are scheduled in the order of their arrivals, thus easily resulting in Head-of-Line (HOL) blocking;
2. *Fair sharing*: Jobs get an equal share of resources, but the execution of early submitted jobs will be prolonged.

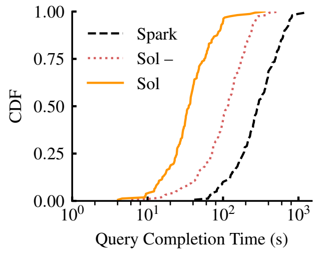
These two schedulers are prevalent in real deployments [1,22], especially when job arrivals and durations are unpredictable.

Improvement of resource utilization Figure 18 shows a timeline of normalized resource usage for both network bandwidth and total CPUs with the FIFO scheduler. A groove in CPU utilization and a peak in network utilization dictate the execution of bandwidth-intensive background jobs. Similarly, a low network utilization but high CPU utilization implicate

[9]We create a 40-node cluster in a single EC2 region.

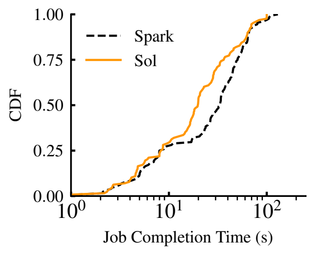


(a) FIFO Scheduler

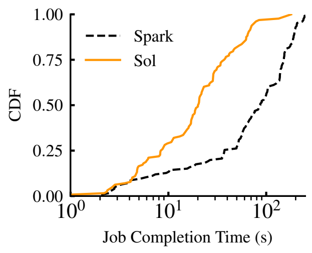


(b) Fair Scheduler

Figure 19: ***JCT with online job arrival using different cluster schedulers. Sol- is Sol without data plane decoupling.***

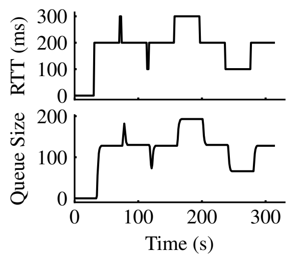


(a) Low Latency and High B/w



(b) Low Latency and Low B/w

Figure 20: ***Sol performance in other design space.***



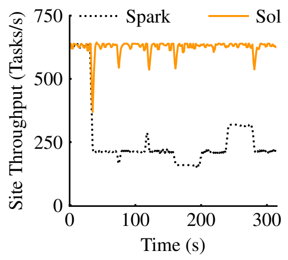


Figure 21: ***Sol performance under latency variations.***

the execution of foreground jobs. We observe Sol improves the CPU utilization by 1.8× over Spark. The source of this improvement comes from both control and data planes: (i) Sol pipelines high-latency coordinations, and thus workers are busy in running tasks all the time. (ii) Sol flexibly repurposes the idle CPU resources in the presence of bandwidth-intensive jobs, thus achieving higher utilizations by orchestrating all jobs. Note that the CPU resource is not always fully saturated in this evaluation, because the cluster is not extremely heavy-loaded given the arrival rate. Therefore, we believe Sol can provide even better performance with heavy workloads, wherein the underutilized resource can be repurposed for more jobs with decoupling. Results were similar for the fair scheduler too.

Improvement of JCTs Figure 19(a) and Figure 19(b) report the distribution of job completion times with FIFO and fair schedulers respectively. The key takeaways are the following. First, simply applying different job schedulers is far from optimal. With the FIFO scheduler, when CloudSort jobs are running, all the frontend jobs are blocked as background jobs hog all the available resources. While the fair scheduler mitigates such job starvation by sharing resource across jobs, it results in a long tail as background jobs are short of resources.

Instead, Sol achieves better job performance by improving both the intra-job and inter-job completions in the task execution level: (i) Early-binding in the control plane improves small jobs, whereby the cluster can finish more jobs in a given time. Hence, even the simple pipelining can achieve an average improvement of 2.6× with the FIFO scheduler and 2.5× with the fair scheduler. (ii) With data plane decoupling, the latency-sensitive jobs can temporarily enjoy under-utilized resource without impacting the bandwidth-intensive jobs. We observe the performance loss of bandwidth-intensive job is less than 0.5%. As the latency-sensitive jobs complete faster, bandwidth-intensive jobs can take up more resource. As such, Sol further improves the average JCTs w.r.t. Spark with both FIFO (average 16.4×) and fair schedulers (average 8.3×).

7.4 Sol's Performance Across the Design Space

We next rerun the prior online experiment to investigate Sol's performance in different network conditions with our cluster.

High bandwidth-low latency network In this evaluation, each machine has 10 Gbps bandwidth, and the latency across machines is <1 ms. Figure 20 shows the distribution of JCTs. The benefits of data plane decoupling depend on the time spent on data exchanges over the network. Although jobs are prone to finishing faster in this favorable environments, Sol can still improve over Spark by 1.3× on average by mitigating the HOL blocking with the decoupling in task executions.

Low bandwidth-low latency network In practice, users may deploy cheap VMs to perform time-insensitive jobs due to budgetary constraints. We now report the JCT distribution in such a setting, where each machine has 1 Gbps low bandwidth and negligible latency. As shown in Figure 20(b), Sol largely outperforms Spark by 3.9×. This gain is again due to the presence of HOL blocking in Spark, where bandwidth-intensive jobs hog their CPUs when tasks are reading large output partitions over the low-bandwidth network.

Note that the high latency-high bandwidth setting rarely exists. As such, Sol can match or achieve noticeable improvement over existing engines across all practical design space.

7.5 Sol's Performance Under Uncertainties

As a network-aware execution engine, Sol can tolerate different uncertainties with its federated design.

Uncertainties in network latency While Sol pushes tasks to site managers with early-binding under high network latency, its performance is robust to latency jitters. We evaluate Sol by continuously feeding our cluster with inference jobs; each scans a 30 GB dataset and the duration of each task is around 100 ms. We snapshot a single site experiencing tran-

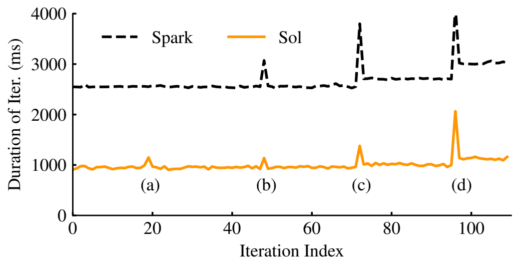


Figure 22: ***Impact of different failures on iteration duration for Sol and Spark: (a) Sol site manager failure, (b) task failure, (c) node failure , and (d) site-wide failure.***

sient or lasting latency variations. As shown in Figure 21, Sol proceeds more tasks than Spark with early-binding of tasks. Moreover, Sol can efficiently react to RTT variations by adaptively tuning its queue size.

Uncertainties in failure Figure 22 compares Sol's performance with Spark under different failures. In this evaluation, we train a long running linear regression in our 10-site deployment, and each iteration performs two stages: training on the data and the aggregation of updates. When the site manager fails (a), Sol restarts the site manager on other local machines, and reschedules the missing queued-up tasks. The recovery of site managers is pipelined with task executions, experiencing little overhead in job performance. Task failures (b) and machine failures (c) in Spark require a tight coordination with the remote coordinator, but Sol handles such failures by coordinating the site manager. Upon detecting task failures, the site manager restarts the task on other locally available machines with its metadata, while asynchronously notifying the coordinator. As such, Sol suffers little overhead by hiding the failures silently. Although the coordinator needs to take charge of rescheduling in both Sol and Spark under site-wide failures (d), tasks in Sol complete faster.

8 Discussion and Future Work

Fine-grained queue management. By capturing the range of task durations, Sol pushes the right number of tasks to site managers at runtime. However, Hoeffding's inequality can be suboptimal, especially when the variance of task durations becomes much greater than their average [23]. Further investigations on the queue management of site managers are needed. To this end, one possible approach is to build a context-aware machine learning model (e.g., reinforcement learning) to decide the optimal queue length [13].

Performance analysis of geo-aware efforts. As the first federated execution engine for diverse network conditions, Sol can serve a large body of existing efforts for geo-distributed data analytics [28, 45, 50]. Although these works do not target latency-bound tasks, for which Sol shows encouraging improvements with control plane optimizations, it would be interesting to investigate Sol's improvement for bandwidth-intensive workloads after applying techniques from existing geo-distributed frameworks.

9 Related Work

Geo-distributed storage and data analytics Numerous efforts strive to build frameworks operating on geo-distributed data. Recent examples include geo-distributed data storage [35, 53] and data analytics frameworks [4, 24]. Geode [51] aims at generating query plans that minimize data transfers over the WAN, while Clarinet [50] and Iridium [45] develop the WAN-aware query optimizer to optimize query response time subject to heterogeneous WAN bandwidth. These optimizations for data analytics lie on the scheduler layer and could transparently leverage Sol for further gains (§6).

Data Processing Engines The explosion of data volumes has fostered the world of MapReduce-based parallel computations [18]. Naiad [40] and Flink [12] express data processing as pipelined fault-tolerant data flows, while the batch processing on them performs similar to Spark [54]. The need for expressive user-defined optimizations motivates Dryad [29] and Apache Tez [47] to enable runtime optimizations on execution plans. These paradigms are designed for well-provisioned networks. Other complementary efforts focus on reasoning about system performance [41, 42], or decoupling communication from computation to further optimize data shuffles [14, 15]. Our work bears some resemblance, but our focus is on designing a network-aware execution engine.

Speeding up data-parallel frameworks Although Nimbus [37] and Drizzle [49] try to speed up execution engines, they focus on amortizing the computation overhead of scheduling for iterative jobs. Hydra [17] democratizes the resource management for jobs across multiple groups. While Yaq-c [46] discusses the tradeoff between utilization and job performance in queue management, its solution is bound to specific task durations without dependencies. Moreover, we optimize task performance inside execution engines.

10 Conclusion

As modern data processing expands due to changing workloads and deployment scenarios, existing execution engines fall short in meeting the requirements of diverse design space. In this paper, we explored the possible designs beyond those for datacenter networks and presented Sol, a federated execution engine that emphasizes an early-binding design in the control plane and decoupling in the data plane. In comparison to the state-of-the-art, Sol can match or achieve noticeable improvements in job performance and resource utilization across all practical points in the design space.

Acknowledgments

Special thanks go to the entire CloudLab team for making Sol experiments possible. We would also like to thank the anonymous reviewers, our shepherd, Ramachandran Ramjee, and SymbioticLab members for their insightful feedback. This work was supported in part by NSF grants CNS-1563095, CNS-1617773, and CNS-1900665.

References

[1] Apache Hadoop NextGen MapReduce (YARN). http://goo.gl/etTGA.

[2] Apache spark. https://spark.apache.org/.

[3] Sort Benchmark. http://sortbenchmark.org/.

[4] TensorFlow Federated. https://www.tensorflow.org/federated.

[5] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds.

[6] TPC Benchmark H (TPC-H). http://www.tpc.org/tpch.

[7] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, and et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[8] G. Ananthanarayanan, A. Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[9] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[10] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. In *EuroSys*, 2018.

[11] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, and et al. Towards federated learning at scale: System design. In *SysML*, 2019.

[12] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[13] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *SIGCOMM*, 2018.

[14] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[15] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3):8, 2013.

[17] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, and et al. Hydra: a federated resource manager for datacenter scale analytics. In *NSDI*, 2019.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[19] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[20] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[21] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[23] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *Journal of the American Statistical Association*, 1963.

[24] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. Ganger, P. Gibbons, and O. Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *NSDI*, 2017.

[25] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[26] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshop*, 2010.

[27] Yuzhen Huang, Yingjie Shi, Zheng Zhong, and et al. Yugong: Geo-Distributed data and job placement at scale. In *VLDB*, 2019.

[28] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *EuroSys*, 2018.

[29] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[30] Anand Padmanabha Iyer, Li Erran Li, Mosharaf Chowdhury, and Ion Stoica. Mitigating the latency-accuracy trade-off in mobile data analytics systems. In *MobiCom*, 2018.

[31] Anand Padmanabha Iyer, Li Erran Li, and Ion Stoica. Celliq : Real-time cellular network analytics at scale. In *NSDI*, 2015.

[32] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 745–761, Carlsbad, CA, October 2018. USENIX Association.

[33] Konstantinos Kloudas, Margarida Mamede, Nuno Preguica, and Rodrigo Rodrigues. Pixida: Optimizing data parallel jobs in wide-area data analytics. In *VLDB*, 2015.

[34] Fan Lai, Mosharaf Chowdhury, and Harsha Madhyastha. To relay or not to relay for inter-cloud transfers? In *HotCloud*, 2018.

[35] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[36] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. Dynamic query re-planning using QOOP. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 253–267, Carlsbad, CA, 2018. USENIX Association.

[37] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *ATC*, 2017.

[38] William Mendenhall, Robert J Beaver, and Barbara M Beaver. *Introduction to probability and statistics*. Cengage Learning, 2012.

[39] Matthew K Mukerjee, David Naylor, Junchen Jiang, Dongsu Han, Srinivasan Seshan, and Hui Zhang. Practical, real-time centralized control for cdn-based live video delivery. In *ACM SIGCOMM Computer Communication Review*, 2015.

[40] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[41] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *SOSP*, 2017.

[42] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.

[43] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[44] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *EuroSys*, 2018.

[45] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Victor Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

[46] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *EuroSys*, 2016.

[47] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curinom. Apache tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*, 2015.

[48] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

[49] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, and Michael J. Franklin. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.

[50] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. Clarinet: WAN-aware optimization for analytics queries. In *OSDI*, 2016.

[51] Ashish Vulimiri, Carlo Curino, B Godfrey, J Padhye, and G Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[52] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[53] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[54] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[55] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[56] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[57] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. Awstream: Adaptive wide-area streaming analytics. In *SIGCOMM*, 2018.

[58] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE S&P*, 2019.

A Benefits of Pipelining

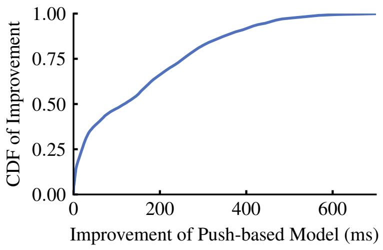


Figure 23: ***Improvement of Push-based Model in (§4.2) .***

To investigate the benefit of pipelining the scheduling and data fetch of a downstream task, we assume zero queuing time and emulate the inter-site coordinations with our measured latency across 44 datacenters on AWS, Azure and Google Cloud. We define the improvement as the difference between the duration of the pull-based model and that of our proposed push-based model for every data fetch (Figure 9). Our results in Figure 23 report we can achieve an average improvement of 153 ms.

Understandably, such benefit is more promising in consideration of the task queuing time, as pushing the remote data is even pipelined with the task spin-wait for scheduling.

B Impact of Queue Length

We quantify the impact of queue size with the aforementioned three workloads (in (§4.2)). In this experiment, we analyze three distinct sites, where the network latency from Site A, B and C to the centralized coordinator is 130 ms, 236 ms and 398 ms, respectively. As shown in Figure 24, queuing up too many or too few tasks can hurt job performance.

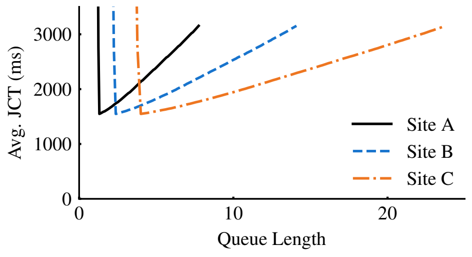
C Determining the Queue Length

Lemma 1. *For a given utilization level* δ *and confidence interval* α *(i.e.,* $\Pr[Util. > \delta] > \alpha$*), the queue length* K *for* S *working slots satisfies:*

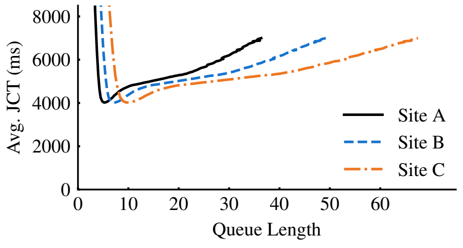
$$K \geq M - \frac{C}{2} + \frac{1}{2D_{avg}}\sqrt{(CD_{avg})^2 - 4MCD_{avg}} \qquad (1)$$

where D_{5th} *and* D_{95th} *denote the 5th and 95th percentile of task durations respectively, and* D_{avg} *denotes the average task duration.* $M = \frac{\delta RTT \times S}{D_{avg}}$, $C = \frac{1}{2}\left(\frac{D_{95th}-D_{5th}}{D_{avg}}\right)^2 \cdot \log\alpha$. *The first term* M *depicts the expectation, while the rest capture the skewness of distributions and confidence.*

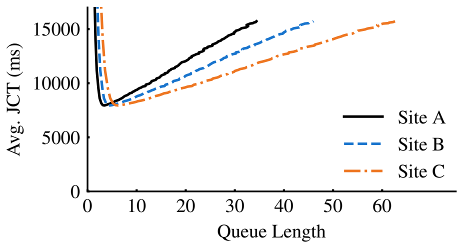
This is true for any distribution of task durations. Unfortunately, we omit the proof for brevity. D_{avg}, D_{5th} and D_{95th} are often stable in a large cluster, and thus available from the historical data. α is the configurable confidence level, which is often set to 99% [32, 38], and δ is set to $\geq$ 100% to guarantee full utilization. Note that from Eq. 1, when task durations follow the uniform distribution, our model ends up with the expectation M. Similarly, when the RTT becomes negligible, this outputs zero queue length.



(a) Uniform Distribution



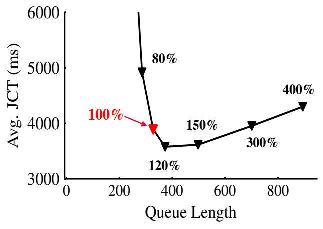
(b) Pareto Distribution



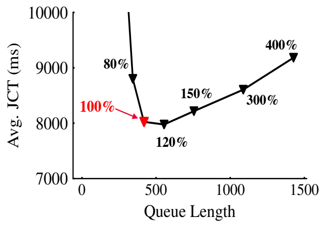
(c) TPC-DS Trace

Figure 24: ***Impact of Queue Size on Job Completion Time (JCT).***

Our evaluations show this model can provide encouraging performance, wherein we reran the prior experiments with the workloads mentioned above (§4.3). We provide the results for workloads with Pareto and TPC-DS distributions by injecting different utilizations in theory, since results for the Uniform distribution are concentrated on a single point (i.e., the expectation M). As shown in Figure 25, the queue length with 100% utilization target locates in the sweet spot of JCTs.



(a) Pareto Distribution



(b) TPC-DS Trace

Figure 25: ***JCT performance with different utilization targets.***

Note that when more task information is available, one can refine this range better; e.g., the bound of Eq. (1) can be improved with Chernoff's inequality when the distribution of task durations is provided.

THEMIS: Fair and Efficient GPU Cluster Scheduling

Kshiteej Mahajan[*], Arjun Balasubramanian[*], Arjun Singhvi[*], Shivaram Venkataraman[*]
Aditya Akella[*], Amar Phanishayee[†], Shuchi Chawla[*]
University of Wisconsin - Madison[*], *Microsoft Research*[†]

Abstract: Modern distributed machine learning (ML) training workloads benefit significantly from leveraging GPUs. However, significant contention ensues when multiple such workloads are run atop a shared cluster of GPUs. A key question is how to fairly apportion GPUs across workloads. We find that established cluster scheduling disciplines are a poor fit because of ML workloads' unique attributes: ML jobs have long-running tasks that need to be gang-scheduled, and their performance is sensitive to tasks' relative placement.

We propose THEMIS, a new scheduling framework for ML training workloads. It's GPU allocation policy enforces that ML workloads complete in *a finish-time fair* manner, a new notion we introduce. To capture placement sensitivity and ensure efficiency, THEMIS uses a two-level scheduling architecture where ML workloads bid on available resources that are offered in an *auction* run by a central arbiter. Our auction design allocates GPUs to winning bids by trading off fairness for efficiency in the short term, but ensuring finish-time fairness in the long term. Our evaluation on a production trace shows that THEMIS can improve fairness by more than $2.25X$ and is ~5% to 250% more cluster efficient in comparison to state-of-the-art schedulers.

1 Introduction

With the widespread success of machine learning (ML) for tasks such as object detection, speech recognition, and machine translation, a number of enterprises are now incorporating ML models into their products. Training individual ML models is time- and resource-intensive with each training job typically executing in parallel on a number of GPUs.

With different groups in the same organization training ML models, it is beneficial to consolidate GPU resources into a shared cluster. Similar to existing clusters used for large scale data analytics, shared GPU clusters for ML have a number of operational advantages, e.g., reduced development overheads, lower costs for maintaining GPUs, etc. However, today, there are no ML workload-specific mechanisms to share a GPU cluster in a *fair* manner.

Our conversations with cluster operators indicate that fairness is crucial; specifically, that sharing an ML cluster becomes attractive to users only if they have the appropriate *sharing incentive*. That is, if there are a total N users sharing a cluster C, every user's performance should be no worse than using a private cluster of size $\frac{C}{N}$. Absent such incentive, users are either forced to sacrifice performance and suffer long wait times for getting their ML jobs scheduled, or abandon shared clusters and deploy their own expensive hardware.

Providing sharing incentive through fair scheduling mechanisms has been widely studied in prior cluster scheduling frameworks, e.g., Quincy [18], DRF [8], and Carbyne [11]. However, these techniques were designed for big data workloads, and while they are used widely to manage GPU clusters today, they are far from effective.

The key reason is that ML workloads have unique characteristics that make existing "fair" allocation schemes actually *unfair*. First, unlike batch analytics workloads, ML jobs have *long running tasks* that need to be scheduled together, i.e., gang-scheduled. Second, each task in a job often runs for a number of iterations while synchronizing model updates at the end of each iteration. This frequent communication means that jobs are *placement-sensitive*, i.e., placing all the tasks for a job on the same machine or the same rack can lead to significant speedups. Equally importantly, as we show, ML jobs differ in their placement-sensitivity (Section 3.1.2).

In Section 3, we show that having long-running tasks means that established schemes such as DRF – which aims to equally allocate the GPUs released upon task completions – can arbitrarily violate sharing incentive. We show that even if GPU resources were released/reallocated on fine time-scales [13], placement sensitivity means that jobs with same aggregate resources could have widely different performance, violating sharing incentive. Finally, heterogeneity in placement sensitivity means that existing scheduling schemes *also violate* Pareto efficiency and envy-freedom, two other properties that are central to fairness [34].

Our scheduler, THEMIS, address these challenges, and supports sharing incentive, Pareto efficiency, and envy-freedom for ML workloads. It multiplexes a GPU cluster across *ML applications* (Section 2), or apps for short, where every app consists of one or more related ML jobs, each running with different hyper-parameters, to train an accurate model for a given task. To capture the effect of long running tasks and placement sensitivity, THEMIS uses a new long-term fairness metric, *finish-time fairness*, which is the ratio of the running time in a shared cluster with N apps to running alone in a $\frac{1}{N}$ cluster. THEMIS's goal is thus to minimize the maximum finish time fairness across all ML apps while efficiently utilizing cluster GPUs. We achieve this goal using two key ideas.

First, we propose to widen the API between ML apps and the scheduler to allow apps to specify placement preferences. We do this by introducing the notion of a round-by-round auction. THEMIS uses leases to account for long-running ML tasks, and auction rounds start when leases expire. At the start of a round, our scheduler requests apps for their finish-time fairness metrics, and makes all available GPUs visible to a fraction of apps that are currently farthest in terms of their fairness metric. Each such app has the opportunity to *bid* for subsets of these GPUs as a part of an auction; bid values reflect the app's new (placement sensitive) finish time fairness metric from acquiring different GPU subsets. A central arbiter determines the global winning bids to maximize the aggregate improvement in the finish time fair metrics across all bidding apps. Using auctions means that we need to ensure that apps are truthful when they bid for GPUs. Thus, we use a *partial allocation* auction that incentivizes truth telling, and ensures Pareto-efficiency and envy-freeness by design.

While a far-from-fair app may lose an auction round, perhaps because it is placed less ideally than another app, its bid values for subsequent auctions naturally increase (because a losing app's finish time fairness worsens), thereby improving the odds of it winning future rounds. Thus, our approach converges to fair allocations over the long term, while staying efficient and placement-sensitive in the short term.

Second, we present a two-level scheduling design that contains a centralized inter-app scheduler at the bottom level, and a narrow API to integrate with existing hyper-parameter tuning frameworks at the top level. A number of existing frameworks such as Hyperdrive [29] and HyperOpt [3] can intelligently apportion GPU resources between various jobs in a single app, and in some cases also terminate a job early if its progress is not promising. Our design allows apps to directly use such existing hyper parameter tuning frameworks. We describe how THEMIS accommodates various hyper-parameter tuning systems and how its API is exercised in extracting relevant inputs from apps when running auctions.

We implement THEMIS atop Apache YARN 3.2.0, and evaluate by replaying workloads from a large enterprise trace. Our results show that THEMIS is at least 2.25X more fair (finish-time fair) than state-of-the-art schedulers while also improving cluster efficiency by ~5% to 250%. To further understand our scheduling decisions, we perform an event-driven simulation using the same trace, and our results show that THEMIS offers greater benefits when we increase the fraction of network intensive apps, and the cluster contention.

2 Motivation

We start by defining the terminology used in the rest of the paper. We then study the unique properties of ML workload traces from a ML training GPU cluster at Microsoft. We end by stating our goals based on our trace analysis and conversations with the cluster operators.

2.1 Preliminaries

We define an ML app, or simply an "app", as a collection of one or more ML model *training* jobs. Each app corresponds to a user training an ML model for a high-level goal, such as speech recognition or object detection. Users train these models knowing the appropriate hyper-parameters (in which case there is just a single job in the app), or they train a closely related set of models (n jobs) that explore hyper-parameters such as learning rate, momentum etc. [21, 29] to identify and train the best target model for the activity at hand.

Each job's constituent work is performed by a number of parallel *tasks*. At any given time, all of a job's tasks collectively process a *mini-batch* of training data; we assume that the size of the batch is fixed for the duration of a job. Each task typically processes a subset of the batch, and, starting from an initial version of the model, executes multiple iterations of the underlying learning algorithm to improve the model. We assume all jobs use the popular synchronous SGD [4].

We consider the finish time of an app to be when the best model and relevant hyper-parameters have been identified. Along the course of identifying such a model, the app may decide to terminate some of its constituent jobs early [3, 29]; such jobs may be exploring hyper-parameters that are clearly sub-optimal (the jobs' validation accuracy improvement over iterations is significantly worse than other jobs in the same app). For apps that contain a single job, finish time is the time taken to train this model to a target accuracy or maximum number of iterations.

2.2 Characterizing Production ML Apps

We perform an analysis of the properties of GPU-based ML training workloads by analyzing workload traces obtained from Microsoft. The GPU cluster we study supports over 5000 unique users. We restrict our analysis to a subset of the trace that contains 85 ML training apps submitted using a hyper-parameter tuning framework.

GPU clusters are known to be heavily contented [19], and we find this also holds true in the subset of the trace of ML apps we consider (Figure 1). For instance, we see that GPU demand is bursty and the average GPU demand is ~50 GPUs.

We also use the trace to provide a first-of-a-kind view into the characteristics of ML apps. As mentioned in Section 2.1, apps may either train a single model to reach a target accuracy (1 job) or may use the cluster to explore various hyper-parameters for a given model (n jobs). Figure 2 shows that ~10% of the apps have 1 job, and around ~90% of the apps perform hyper-parameter exploration with as many as 100 jobs (median of 75 jobs). Interestingly, there is also a significant variation in the number of hyper-parameters explored ranging from a few tens to about a hundred (not shown).

We also measure the *GPU time* of all ML apps in the trace. If an app uses 2 jobs with 2 GPUs each for a period of 10 minutes, then the GPU time for — the tasks would be 10 minutes each, the jobs would be 20 minutes each, and the app would be 40 GPU minutes. Figure 3 and Figure 4 show the long running nature of ML apps: the median app takes 11.5 GPU days and the median task takes 3.75 GPU hours. There is a wide diversity with a significant fraction of jobs and apps that are more than 10X shorter and many that are more than 10X longer.

From our analysis we see that ML apps are heterogeneous in terms of resource usage, and number of jobs submitted.

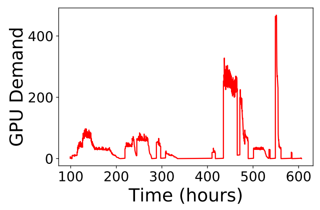


Figure 1: Aggregate GPU demand of ML apps over time

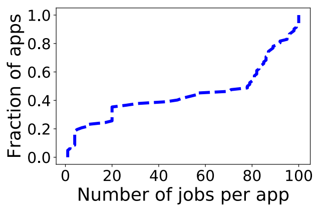


Figure 2: Job count distribution across different apps

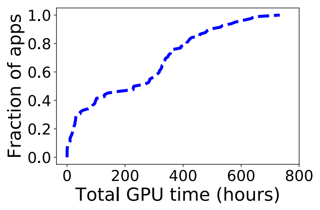


Figure 3: ML app time (= total GPU time across all jobs in app) distribution

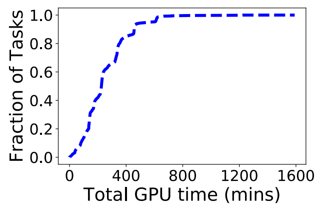


Figure 4: Distribution of Task GPU times

Running times are also heterogeneous, but at the same time much longer than, e.g., running times of big data analytics jobs (typically a few hours [12]). Handling such heterogeneity can be challenging for scheduling frameworks, and the long running nature may make controlling app performance particularly difficult in a shared setting with high contention.

We next discuss how some of these challenges manifest in practice from both cluster user and cluster operator perspectives, and how that leads to our design goals for THEMIS.

2.3 Our Goal

Our many conversations with operators of GPU clusters revealed a common sentiment, reflected in the following quote:

" We were scheduling with a balanced approach ... with guidance to 'play nice'. Without firm guard rails, however, there were always individuals who would ignore the rules and dominate the capacity. "

— An operator of a large GPU cluster at Microsoft

With long app durations, users who dominate capacity impose high waiting times on many other users. Some such users are forced to "quit" the cluster as reflected in this quote:

"Even with existing fair sharing schemes, we do find users frustrated with the inability to get their work done in a timely way... The frustration frequently reaches the point where groups attempt or succeed at buying their own hardware tailored to their needs. "

— An operator of a large GPU cluster at Microsoft

While it is important to design a cluster scheduler that ensures efficient use of highly contended GPU resources, the above indicates that it is perhaps equally, if not more important, for the scheduler to allocate GPU resources in a fair manner across many diverse ML apps; in other words, roughly speaking, the scheduler's goal should be to allow all apps to execute their work in a "timely way".

In what follows, we explain using examples, measurements, and analysis, why existing fair sharing approaches when applied to ML clusters fall short of the above goal, which we formalize next. We identify the need both for a new fairness metric, and for a new scheduler architecture and API that supports resource division according to the metric.

3 Finish-Time Fair Allocation

We present additional unique attributes of ML apps and discuss how they, and the above attributes, affect existing fair sharing schemes.

3.1 Fair Sharing Concerns for ML Apps

The central question is - given R GPUs in a cluster C and N ML apps, what is a *fair way* to divide the GPUs.

As mentioned above, cluster operators indicate that the primary concern for users sharing an ML cluster is performance isolation that results in "timely completion". We formalize this as: if N ML Apps are sharing a cluster then an app should not run slower on the shared cluster compared to a dedicated cluster with $\frac{1}{N}$ of the resources. Similar to prior work [8], we refer to this property as *sharing incentive* (SI). Ensuring sharing incentive for ML apps is our primary design goal.

In addition, resource allocation mechanisms must satisfy two other basic properties that are central to fairness [34]: Pareto Efficiency (PE) and Envy-Freeness (EF) [1]

While prior systems like Quincy [18], DRF [8] etc. aim at providing SI, PE and EF, we find that they are ineffective for ML clusters as they fail to consider *the long durations of ML tasks* and *placement preferences of ML apps*.

3.1.1 ML Task Durations

We empirically study task durations in ML apps and show how they affect the applicability of existing fair sharing schemes.

Figure 4 shows the distribution of task durations for ML apps in a large GPU cluster at Microsoft. We note that the tasks are, in general, very long, with the median task roughly 3.75 hours long. This is in stark contrast with, e.g., big data analytics jobs, where tasks are typically much shorter in duration [26].

State of the art fair allocation schemes such as DRF [8] provide instantaneous resource fairness. Whenever resources become available, they are allocated to the task from an app with the least current share. For big data analytics, where task durations are short, this approximates instantaneous resource fairness, as frequent task completions serve as opportunities to redistribute resources. However, blindly applying such schemes to ML apps can be disastrous: running the much longer-duration ML tasks to completion could lead to newly arriving jobs waiting inordinately long for resources. This leads to violation of SI for late-arriving jobs.

Recent "attained-service" based schemes address this problem with DRF. In [13], for example, GPUs are leased for a certain duration, and when leases expire, available GPUs are given to the job that received the least GPU time thus far;

[1]Informally, a Pareto Efficient allocation is one where no app's allocation can be improved without hurting some other app. And, envy-freeness means that no app should prefer the resource allocation of an other app.

	VGG16	Inception-v3
4 P100 GPUs on 1 server	103.6 images/sec	242 images/sec
4 P100 GPUs across 2 servers	80.4 images/sec	243 images/sec

Table 1: Effect of GPU resource allocation on job throughput. VGG16 has a machine-local task placement preference while Inception-v3 does not.

this is the "least attained service", or LAS allocation policy. While this scheme avoids the starvation problem above for late-arriving jobs, it still violates all key fairness properties because it is placement-unaware, an issue we discuss next.

3.1.2 Placement Preferences

Next, we empirically study placement preferences of ML apps. We use examples to show how ignoring these preferences in fair sharing schemes violates key properties of fairness.

Many apps, many preference patterns: ML cluster users today train a variety of ML apps across domains like computer vision, NLP and speech recognition. These models have significantly different model architectures, and more importantly, different placement preferences arising from different computation, communication needs. For example, as shown in Table 1, VGG16 has a strict machine-local task placement preference while Inception-v3 does not. This preference inherently stems from the fact that VGG-like architectures have very large number of parameters and incur greater overheads for updating gradients over the network.

We use examples to show the effect of placement on DRF's allocation strategy. Similar examples and conclusions apply for the LAS allocation scheme.

Ignoring placement affects SI: example: Consider the Instance 1 in Figure 5. In this example, there are two placement sensitive ML apps - A_1 and A_2, both training VGG16. Each ML app has just one job in it with 4 tasks and the cluster has two 4 GPU machines. As shown above, given the same number of GPUs both apps prefer GPUs to be in the same server than spread across servers.

For this example, DRF [8] equalizes the dominant resource share of both the apps under resource constraints and allocates 4 GPUs to each ML app. In Instance 1 of Figure 5 we show an example of a valid DRF allocation. Both apps get the same type of placement with GPUs spread across servers. This allocation violates SI for both apps as their performance would be better if each app just had its own dedicated server.

Ignoring placement affects PE, EF: example: Consider Instance 2 in Figure 5 with two apps - A_1 (Inception-v3) which is not placement sensitive and A_2 (VGG16) which is placement sensitive. Each app has one job with four tasks and the cluster has two machines: one 4 GPU and two 2 GPU.

Now consider the allocation in Instance 2, where A_1 is allocated on the 4 GPU machine whereas A_2 is allocated across the 2 GPU machines. This allocation violates EF, because A_2 would prefer A_1's allocation. It also violates PE because swapping the two apps' allocation would improve A_2's performance without hurting A_1.

In fact, we can formally show that:

Theorem 3.1. Existing fair schemes (DRF, LAS) ignore placement preferences and violate SI, PE, EF for ML apps.

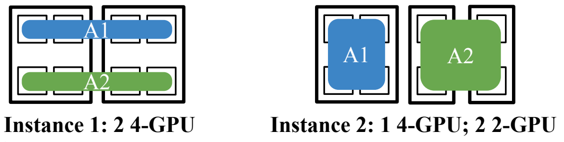


Figure 5: By ignoring placement preference, DRF violates sharing incentive.

$\vec{G}$	$[0,0]$	$[1,0] = [1,0]$	$[1,1]$
ρ	ρ_{old}	$\frac{200}{400} = \frac{1}{2}$	$\frac{100}{400} = \frac{1}{4}$

Table 2: Example table of bids sent from apps to the scheduler

Proof Refer to Appendix.

In summary, existing schemes fail to provide fair sharing guarantees as they are unaware of ML app characteristics. Instantaneous fair schemes such as DRF fail to account for long task durations. While least-attained service schemes overcome that limitation, neither approach's input encodes placement preferences. Correspondingly, the fairness metrics used - i.e., dominant resource share (DRF) or attained service (LAS) - do not capture placement preferences.

This motivates the need for a new placement-aware fairness metric, and corresponding scheduling discipline. Our observations about ML task durations imply that, like LAS, our fair allocation discipline should not depend on rapid task completions, but instead should operate over longer time scales.

3.2 Metric: Finish-Time Fairness

We propose a new metric called as finish-time fairness, ρ. $\rho = \frac{T_{sh}}{T_{id}}$.

T_{id} is the *independent finish-time* and T_{sh} is the *shared finish-time*. T_{sh} is the finish-time of the app in the shared cluster and it encompasses the slowdown due to the placement and any queuing delays that an app experiences in getting scheduled in the shared cluster. The worse the placement, the higher is the value of T_{sh}. T_{id}, is the finish-time of the ML app in its own independent and exclusive $\frac{1}{N}$ share of the cluster. Given the above definition, sharing incentive for an ML app can be attained if $\rho \leq 1$. [2]

To ensure this, it is necessary for the allocation mechanism to estimate the values of ρ for different GPU allocations. Given the difficulty in predicting how various apps will react to different allocations, it is intractable for the scheduling engine to predict or determine the values of ρ.

Thus, we propose a new wider interface between the app and the scheduling engine that can allow the app to express a *preference* for each allocation. We propose that apps can encode this information as a table. In Table 2, each column has a permutation of a potential GPU allocation and the estimate of ρ on receiving this allocation. We next describe how the scheduling engine can use this to provide fair allocations.

3.3 Mechanism: Partial Allocation Auctions

The finish-time fairness $\rho_i(.)$ for an ML app A_i is a function of the GPU allocation $\vec{G_i}$ that it receives. The allocation policy

[2]Note, sharing incentive criteria of $\rho \leq 1$ assumes the presence of an admission control mechanism to limit contention for GPU resources. An admission control mechanism that rejects any app if the aggregate number of GPUs requested crosses a certain threshold is a reasonable choice.

Pseudocode 1 Finish-Time Fair Policy

```
 1: Applications $\{A_i\}$                                   ▷ set of apps
 2: Bids $\{\rho_i(.)\}$                        ▷ valuation function for each app $i$
 3: Resources $\overrightarrow{R}$                    ▷ resource set available for auction
 4: Resource Allocations $\{\overrightarrow{G}_i\}$        ▷ resource allocation for each app $i$

 5: procedure AUCTION($\{A_i\}$, $\{\rho_i(.)\}$, $\overrightarrow{R}$)
 6:     $\overrightarrow{G}_{i,pf} = \arg\max \prod_i 1/\rho_i(\overrightarrow{G_i})$   ▷ proportional fair (pf) allocation per app $i$
 7:     $\overrightarrow{G}^{-i}_{j,pf} = \arg\max \prod_{j!=i} 1/\rho_j(\overrightarrow{G_j})$   ▷ pf allocation per app $j$ without app $i$
 8:     $c_i = \frac{\prod_{j!=i} 1/\rho_j(\overrightarrow{G}_{j,pf})}{\prod_{j!=i} 1/\rho_j(\overrightarrow{G}^{-i}_{j,pf})}$
 9:     $\overrightarrow{G_i} = c_i * \overrightarrow{G}_{i,pf}$                        ▷ allocation per app $i$
10:     $\overrightarrow{L} = \sum_i 1 - c_i * \overrightarrow{G}_{i,pf}$              ▷ aggregate leftover resource
11:     return $\{\overrightarrow{G_i}\}$, $\overrightarrow{L}$
12: end procedure
13: procedure ROUNDBYROUNDAUCTIONS($\{A_i\}$, $\{\rho_i(.)\}$)
14:     while True do
15:         ONRESOURCEAVAILABLEEVENT $\vec{R'}$:
16:         $\{A_i^{sort}\}$ = SORT($\{A_i\}$) on $\rho_i^{current}$
17:         $\{A_i^{filter}\}$ = get top $1 - f$ fraction of apps from $\{A_{sort}\}$
18:         $\{\rho_i^{filter}(.)\}$ = get updated $\rho(.)$ from apps in $\{A_i^{filter}\}$
19:         $\{\overrightarrow{G_i^{filter}}\}$, $\overrightarrow{L}$ = AUCTION($\{A_i^{filter}\}$, $\{\rho_i^{filter}(.)\}$, $\vec{R'}$)
20:         $\{A_i^{unfilter}\} = \{A_i\} - \{A_i^{filter}\}$
21:         allocate $\overrightarrow{L}$ to $\{A_i^{unfilter}\}$ at random
22:     end while
23: end procedure
```

takes these $\rho_i(.)$'s as inputs and outputs allocations $\vec{G_i}$.

A straw-man policy that sorts apps based on their reported ρ_i values and allocates GPUs in that order reduces the maximum value of ρ but has one key issue. An app can submit *false information* about their ρ values. This greedy behavior can boost their chance of winning allocations. Our conversations with cluster operators indicate that apps request for more resources than required and they require manual monitoring ("*We also monitor the usage. If they don't use it, we reclaim it and pass it on to the next approved project*"). Thus, this simple straw-man fails to incentivize truth-telling and violates another key property, namely, *strategy proofness* (SP).

To address this challenge, we propose to use *auctions* in THEMIS. We begin by describing a simple mechanism that runs a single-round auction and then extend to a round-by-round mechanism that also considers online updates.

3.3.1 One-Shot Auction

Details of the inputs necessary to run the auction are given first, followed by how the auction works given these inputs.

Inputs: Resources and Bids. $\vec{R}$ represents the total GPU resources to be auctioned, where each element is 1 and the number of dimensions is the number of GPUs to be auctioned.

Each ML app bids for these resources. The bid for each ML app consists of the estimated finish-time fair metric (ρ_i) values for several different GPU allocations ($\vec{G_i}$). Each element in $\vec{G_i}$ can be $\{0,1\}$. A set bit implies that GPU is allocated to the app. Example of a bid can be seen in Table 2.

Auction Overview. To ensure that the auction can provide strategy proofness, we propose using a *partial allocation* auction (PA) mechanism [5]. Partial allocation auctions have been shown to incentivize truth telling and are an appropriate fit for modeling subsets of indivisible goods to be auctioned across apps. Pseudocode 1, line 5 shows the PA mechanism. There are two aspects to auctions that are described next.

1. Initial allocation. PA starts by calculating an intrinsically proportionally-fair allocation $\overrightarrow{G_{i,pf}}$ for each app A_i by maximizing the product of the valuation functions i.e., the finish-time fair metric values for all apps (Pseudocode 1, line 6). Such an allocation ensures that it is not possible to increase the allocation of an app without decreasing the allocation of at least another app (satisfying PE [5]).

2. Incentivizing Truth Telling. To induce truthful reporting of the bids, the PA mechanism allocates app A_i only a fraction $c_i < 1$ of A_i's proportional fair allocation $\overrightarrow{G_{i,pf}}$, and takes $1 - c_i$ as a *hidden payment* (Pseudocode 1, line 10). The c_i is directly proportional to the decrease in the collective valuation of the other bidding apps in a market with and without app A_i (Pseudocode 1, line 8). This yields the final allocation $\vec{G_i}$ for app A_i (Pseudocode 1, line 9).

Note that the final result, $\vec{G_i}$ is not a market-clearing allocation and there could be unallocated GPUs $\vec{L}$ that are leftover from hidden payments. Hence, PA is not work-conserving. Thus, while the one-shot auction provides a number of properties related to fair sharing it does not ensure SI is met.

Theorem 3.2. The one-shot partial allocation auction guarantees SP, PE and EF, but does not provide SI.

Proof Refer to Appendix. The intuitive reason for this is that, with unallocated GPUs as hidden payments, PA does not guarantee $\rho \leq 1$ for all apps. To address this we next look at multi-round auctions that can maximize SI for ML apps. We design a mechanism that is based on PA and preserves its properties, but offers slightly weaker guarantee, namely min max ρ. We describe this next. It runs in multiple rounds. Empirically, we find that it gets $\rho \leq 1$ for most apps, even without admission control.

3.3.2 Multi-round auctions

To maximize sharing incentive and to ensure work conservation, our goal is to ensure $\rho \leq 1$ for as many apps as possible. We do this using three key ideas described below.

Round-by-Round Auctions: With round-by-round auctions, the outcome of an allocation from an auction is binding only for a *lease* duration. At the end of this lease, the freed GPUs are re-auctioned. This also handles the online case as any auction is triggered on a *resource available event*. This takes care of app failures and arrivals, as well as cluster reconfigurations.

At the beginning of each round of auction, the policy solicits updated valuation functions $\rho(.)$ from the apps. The estimated work and the placement preferences for the case of ML apps are typically time varying. This also makes our policy adaptive to such changes.

Round-by-Round Filtering: To maximize the number of apps with $\rho \leq 1$, at the beginning of each round of auctions we filter the $1 - f$ fraction of total active apps with the greatest values of current estimate of their finish-time fair metric ρ. Here, $f \in (0,1)$ is a system-wide parameter.

This has the effect of restricting the auctions to the apps that are at risk of not meeting SI. Also, this restricts the auction to a smaller set of apps which reduces contention for resources and hence results in smaller hidden payments. It also makes the auction computationally tractable.

Over the course of many rounds, filtering maximizes the number of apps that have SI. Consider a far-from-fair app i that lost an auction round. It will appear in future rounds with much greater likelihood relative to another less far-from-fair app k that won the auction round. This is because, the winning app k was allocated resources; as a result, it will see its ρ improve over time; thus, it will eventually not appear in the fraction $1-f$ of not-so-fairly-treated apps that participate in future rounds. In contrast, i's ρ will increase due to the waiting time, and thus it will continue to appear in future rounds. Further an app that loses multiple rounds will eventually lose its lease on all resources and make no further progress, causing its ρ to become unbounded. The next auction round the app participates in will likely see the app's bid winning, because any non-zero GPU allocation to that app will lead to a huge improvement in the app's valuation.

As $f \rightarrow 1$, our policy provides greater guarantee on SI. However, this increase in SI comes at the cost of efficiency. This is because $f \rightarrow 1$ restricts the set of apps to which available GPUs will be allocated; with $f \rightarrow 0$ available GPUs can be allocated to apps that benefit most from better placement, which improves efficiency at the risk of violating SI.

Leftover Allocation: At the end of each round we have leftover GPUs due to hidden payments. We allocate these GPUs at random to the apps that did not participate in the auction in this round. Thus our overall scheme is work-conserving.

Overall, we prove that:

Theorem 3.3. Round-by-round auctions preserve the PE, EF and SP properties of partial auctions and maximize SI.

*P*roof. Refer to Appendix.

To summarize, in THEMIS we propose a new finish-time fairness metric that captures fairness for long-running, placement sensitive ML apps. To perform allocations, we propose using a multi-round partial allocation auction that incentivizes truth telling and provides Pareto efficient, envy free allocations. By filtering the apps considered in the auction, we maximize sharing incentive and hence satisfy all the properties necessary for fair sharing among ML applications.

4 System Design

We first list design requirements for an ML cluster scheduler taking into account the fairness metric and auction mechanism described in Section 3, and the implications for the THEMIS scheduler architecture. Then, we discuss the *API* between the scheduler and the hyper-parameter optimizers.

4.1 Design Requirements

Separation of visibility and allocation of resources. Core to our partial allocation mechanism is the abstraction of making available resources visible to a number of apps but allocating each resource exclusively to a single app. As we argue below, existing scheduling architectures couple these concerns and thus necessitate the design of a new scheduler.

Integration with hyper-parameter tuning systems. Hyper-parameter optimization systems such as Hyperband [21], Hyperdrive [29] have their own schedulers that decide the resource allocation and execution schedule for the jobs within those apps. We refer to these as app-schedulers. One of our goals in THEMIS is to integrate with these systems with minimal modifications to app-schedulers.

These two requirements guide our design of a new *two-level semi-optimistic* scheduler and a set of corresponding abstractions to support hyper-parameter tuning systems.

4.2 THEMIS Scheduler Architecture

Existing scheduler architectures are either pessimistic or fully optimistic and both these approaches are not suitable for realizing multi-round auctions. We first describe their shortcomings and then describe our proposed architecture.

4.2.1 Need for a new scheduling architecture

Two-level pessimistic schedulers like Mesos [17] enforce pessimistic concurrency control. This means that visibility and allocation go hand-in-hand at the granularity of a single app. There is restricted single-app visibility as available resources are partitioned by a mechanism internal to the lower-level (i.e., cross-app) scheduler and offered only to a single app at a time. The tight coupling of visibility and allocation makes it infeasible to realize round-by-round auctions where resources need to be visible to many apps but allocated to just one app.

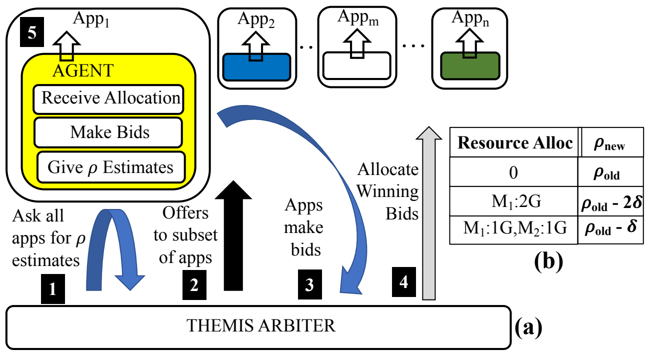
Shared-state fully optimistic schedulers like Omega [30] enforce fully optimistic concurrency control. This means that visibility and allocation go hand-in-hand at the granularity of multiple apps. There is full multi-app visibility as all cluster resources and their state is made visible to all apps. Also, all apps contend for resources and resource allocation decisions are made by multiple apps at the same time using transactions. This coupling of visibility and allocation in a lock-free manner makes it hard to realize a global policy like finish-time fairness and also leads to expensive conflict resolution (needed when multiple apps contend for the same resource) when the cluster is highly contented, which is typically the case in shared GPU clusters.

Thus, the properties required by multi-round auctions, i.e., multi-app resource visibility and single-app resource allocation granularity, makes existing architectures ineffective.

4.2.2 Two-Level Semi-Optimistic Scheduling

The two-levels in our scheduling architecture comprise of multiple app-schedulers and a cross-app scheduler that we call the ARBITER. The ARBITER has our scheduling logic. The top level per-app schedulers are minimally modified to interact with the ARBITER. Figure 6 shows our architecture.

Each GPU in a THEMIS-managed cluster has a lease associated with it. The lease decides the duration of ownership of the GPU for an app. When a lease expires, the resource is made



Resource Alloc	ρ_{new}
0	ρ_{old}
M$_1$:2G	ρ_{old} - 2δ
M$_1$:1G,M$_2$:1G	ρ_{old} - δ

Figure 6: THEMIS Design. (a) Sequence of events in THEMIS - starts with a resource available event and ends with resource allocations. (b) Shows a typical bid valuation table an App submits to ARBITER. Each row has a subset of the complete resource allocation and the improved value of ρ_{new}.

available for allocation. THEMIS's ARBITER pools available resources and runs a round of the auctions described earlier. During each such round, the resource allocation proceeds in 5 steps spanning 2 phases (shown in Figure 6):

The first phase, called the *visibility phase*, spans steps 1–3.

1 The ARBITER asks all apps for current finish-time fair metric estimates. **2** The ARBITER initiates auctions, and makes the same non-binding resource-offer of the available resources to a fraction $f \in [0,1]$ of ML apps with worst finish-time fair metrics (according to round-by-round filtering described earlier). To minimize changes in the ML app scheduler to participate in auctions, THEMIS introduces an AGENT that is co-located with each ML app scheduler. The AGENT serves as an intermediary between the ML app and the ARBITER. **3** The apps examine the resource offer in parallel. Each app's AGENT then replies with a single bid that contains preferences for desired resource allocations.

The second phase, *allocation phase*, spans steps 4–5. **4** The ARBITER, upon receiving all the bids for this round, picks winning bids according to previously described partial allocation algorithm and leftover allocation scheme. It then notifies each AGENT of its winning allocation (if any). **5** The AGENT propagates the allocation to the ML app scheduler, which can then decide the allocation among constituent jobs.

In sum, the two phase resource allocation means that our scheduler enforces *semi-optimistic concurrency control*. Similar to fully optimistic concurrency control, there is multi-app visibility as the cross-app scheduler offers resources to multiple apps concurrently. At the same time, similar to pessimistic concurrency control, the resource allocations are conflict-free guaranteeing exclusive access of a resource to every app.

To enable preparation of bids in step **3**, THEMIS implements a narrow API from the ML app scheduler to the AGENT that enables propagation of app-specific information. An AGENT's bid contains a *valuation function* ($\rho(.)$) that provides, for each resource subset, an estimate of the finish-time fair metric the app will achieve with the allocation of the resource subset. We describe how this is calculated next.

4.3 AGENT and AppScheduler Interaction

An AGENT co-resides with an app to aid participation in auctions. We now describe how AGENTs prepare bids based on inputs provided by apps, the API between an AGENT and its app, and how AGENTs integrate with current hyper-parameter optimization schedulers.

4.3.1 Single-Job ML Apps

For ease of explanation, we first start with the simple case of an ML app that has just one ML training job which can use at most job_demand_{max} GPUs. We first look at calculation of the finish-time fair metric, ρ. We then look at a multi-job app example so as to better understand the various steps and interfaces in our system involved in a multi-round auction.

Calculating $\rho(\vec{G})$. Equation 1 shows the steps for calculating ρ for a single job given a GPU allocation of $\vec{G}$ in a cluster C with R_C GPUs. When calculating ρ we assume that the allocation $\vec{G}$ is binding till job completion.

$$\begin{aligned}
\rho(\vec{G}) &= T_{sh}(\vec{G})/T_{id} \\
T_{sh} &= T_{current} - T_{start} + \\
&\quad iter\_left * iter\_time(\vec{G}) \\
T_{id} &= T_{cluster} * N_{avg} \\
iter\_time(\vec{G}) &= \frac{iter\_time\_serial * \mathcal{S}(\vec{G})}{min(||\vec{G}||_1, job\_demand_{max})} \\
T_{cluster} &= \frac{iter\_total * iter\_serial\_time}{min(R_C, job\_demand_{max})}
\end{aligned} \tag{1}$$

T_{sh} is the shared finish-time and is a function of the allocation $\vec{G}$ that the job receives. For the single job case, it has two terms. First, is the time elapsed ($= T_{current} - T_{start}$). Time elapsed also captures any queuing delays or starvation time. Second, is the time to execute remaining iterations which is the product of the number of iterations left ($iter\_left$) and the iteration time ($iter\_time(\vec{G})$). $iter\_time(\vec{G})$ depends on the allocation received. Here, we consider the common-case of the ML training job executing synchronous SGD. In synchronous SGD, the work in an iteration can be parallelized across multiple workers. Assuming linear speedup, this means that the iteration time is the serial iteration time ($iter\_time\_serial$) reduced by a factor of the number of GPUs in the allocation, $||\vec{G}||_1$ or job_demand_{max} whichever is lesser. However, the linear speedup assumption is not true in the common case as network overheads are involved. We capture this via a slowdown penalty, $\mathcal{S}(\vec{G})$, which depends on the placement of the GPUs in the allocation. Values for $\mathcal{S}(\vec{G})$ can typically be obtained by profiling the job offline for a few iterations. [3] The slowdown is captured as a multiplicative factor, $\mathcal{S}(\vec{G}) \geq 1$, by which T_{sh} is increased.

[3] $\mathcal{S}(\vec{G})$ can also be calculated in an online fashion. First, we use crude placement preference estimates to begin with for single machine (=1), cross-machine (=1.1), cross-rack (=1.3) placement. These are replaced with accurate estimates by profiling iteration times when the ARBITER allocates

T_{id} is the estimated finish-time in an independent $\frac{1}{N_{avg}}$ cluster. N_{avg} is the average contention in the cluster and is the weighted average of the number of apps in the system during the lifetime of the app. We approximate this as the finish-time of the app in the whole cluster, $T_{cluster}$ multiplied by the average contention. $T_{cluster}$ assumes linear speedup when the app executes with all the cluster resources R_C or maximum app demand whichever is lesser. It also assumes no slowdown. Thus, it is approximated as $\frac{iter\_total*iter\_serial\_time}{min(R_C, job\_demand_{max})}$.

4.3.2 Generalizing to Multiple-Job ML Apps

ML app schedulers for hyper-parameter optimization systems typically go from aggressive exploration of hyper-parameters to aggressive exploitation of best hyper-parameters. While there are a number of different algorithms for choosing the best hyper-parameters [3, 21] to run, we focus on early stopping criteria as this affects the finish time of ML apps.

As described in prior work [9], automatic stopping algorithms can be divided into two categories: Successive Halving and Performance Curve Stopping. We next discuss how to compute T_{sh} for each case.

Successive Halving refers to schemes which start with a total time or iteration budget B and apportion that budget by periodically stopping jobs that are not promising. For example, if we start with n hyper parameter options, then each one is submitted as a job with a demand of 1 GPU for a fixed number of iterations I. After I iterations, only the best $\frac{n}{2}$ ML training jobs are retained and assigned a maximum demand of 2 GPUs for the same number of iterations I. This continues until we are left with 1 job with a maximum demand of n GPUs. Thus there are a total of $log_2 n$ *phases* in Successive Halving. This scheme is used in Hyperband [21] and Google Vizier [9].

We next describe how to compute T_{sh} and T_{id} for successive halving. We assume that the given allocation $\overrightarrow{G}$ lasts till app completion and the total time can be computed by adding up the time the app spends for each phase. Consider the case of phase i which has $J = \frac{n}{2^{i-1}}$ jobs. Equation 2 shows the calculation of $T_{sh(i)}$, the shared finish time of the phase. We assume a separation of concerns where the hyper-parameter optimizer can determine the optimal allocation of GPUs *within a phase* and thus estimate the value of $\mathcal{S}(\overrightarrow{G_j})$. Along with *iter_left*, *serial_iter_time*, the AGENT can now estimate $T_{sh(j)}$ for each job in the phase. We mark the phase as finished when the slowest or last job in the app finishes the phase (max_j). Then the shared finish time for the app is the sum of the finish times of all constituent phases.

To estimate the ideal finish-time we compute the total time to execute the app on the full cluster. We estimate this using the budget B which represents the aggregate work to be done and, as before, we assume linear speedup to the maximum number of GPUs the app can use app_demand_{max}.

unseen placements. The multi-round nature of allocations means that errors in early estimates do not have a significant effect.

$$
\begin{aligned}
T_{sh(i)} &= max_j\{T(\overrightarrow{G_j})\}\\
T_{sh} &= \sum_i T_{sh(i)}\\
T_{cluster} &= \frac{B}{min(R_C, app\_demand_{max})}\\
T_{id} &= T_{cluster} * N_{avg}
\end{aligned}
\tag{2}
$$

The AGENT generates ρ using the above procedure for all possible subsets of $\{\overrightarrow{G}\}$ and produces a bid table similar to the one shown in Table 2 before. The API between the AGENT and hyper-parameter optimizer is shown in Figure 7 and captures the functions that need to implemented by the hyper-parameter optimizer.

Performance Curve Stopping refers to schemes where the convergence curve of a job is extrapolated to determine which jobs are more promising. This scheme is used by Hyperdrive [29] and Google Vizier [9]. Computing T_{sh} proceeds by calculating the finish time for each job that is currently running by estimating the iteration at which the job will be terminated (thus T_{sh} is determined by the job that finishes last). As before, we assume that the given allocation $\overrightarrow{G}$ lasts till app completion. Since the estimations are usually probabilistic, i.e., the iterations at which the job will converge has an error bar, we over-estimate and use the most optimistic convergence curve that results in the maximum forecasted completion time for that job. As the job progresses, the estimates of the convergence curve get more accurate and improves the accuracy of the estimated finish time T_{sh}. The API implemented by the hyper-parameter optimizer is simpler and only involves getting a list of running jobs as shown in Figure 7.

We next present an end-to-end example of a multi-job app showing our mechanism in action.

4.3.3 End-to-end Example.

We now run through a simple example that exercises the various aspects of our API and the interfaces involved.

Consider a 16 GPU cluster and an ML app that has 4 ML jobs and uses successive halving, running along with 3 other ML apps in the same cluster. Each job in the app is tuning a different hyper-parameter and the serial time taken per iteration for the jobs are $80, 100, 100, 120$ seconds respectively.[4] The total budget for the app is $10,000$ seconds of GPU time and we assume the job_demand_{max} is 8 GPUs and $\mathcal{S}(\overrightarrow{G}) = 1$.

Given we start with 4 ML jobs, the hyper-parameter optimizer divides this into 3 phases each having $4, 2, 1$ jobs, respectively. To evenly divide the budget across the phases, the hyper-parameter optimizer budgets $\approx 8, 16, 36$ iterations in each phase. First we calculate the T_{id} by considering the budget, total cluster size, and cluster contention as: $\frac{10000\times4}{16} =$ 2500s.

Next, we consider the computation of T_{sh} assuming that 16

[4]The time per iteration depends on the nature of the hyper-parameter being tuned. Some hyper-parameters like batch size or quantization used affect the iteration time while others like learning rate don't.

```
class JobInfo(int itersRemaining,
              float avgTimePerIter,
              float localitySensitivity);
// Successive Halving
List<JobInfo> getJobsInPhase(int phase,
                             List<Int> gpuAlloc);
int getNumPhases();
// Performance Curve
List<JobInfo> getJobsRemaining(List<Int> gpuAlloc);
```

Figure 7: API between AGENT and hyperparameter optimizer

$\|\|\vec{G}\|\|_1$	0	1	2	4	8	16
ρ	ρ_{old}	4	2	1	0.5	0.34

Table 3: Example of bids submitted by AGENT

GPUs are offered by the ARBITER. The AGENT now computes the bid for each subset of GPUs offered. Consider the case with 2 GPUs. In this case in the first phase we have 4 jobs which are serialized to run 2 at a time. This leads to $T_{sh(1)} = (120 \times 8) + (80 \times 8) = 1600$ seconds. (Assume two 100s jobs run serially on one GPU, and the 80 and 120s jobs run serially on the other. T_{sh} is the time when the last job finishes.)

When we consider the next stage the hyper-parameter optimizer currently does not know which jobs will be chosen for termination. We use the *median* job (in terms of per-iteration time) to estimate $T_{sh(i)}$ for future phases. Thus, in the second phase we have 2 jobs so we run one job on each GPU each of which we assume to take the median 100 seconds per iteration leading to $T_{sh(2)} = (100 \times 16) = 1600$ seconds. Finally for the last phase we have 1 job that uses 2 GPUs and runs for 36 iterations leading to $T_{sh(3)} = \frac{(100\times36)}{2} = 1800$ (again, the "median" jobs takes 100s per iteration). Thus $T_{sh} = 1600 + 1600 + 1800 = 5000$ seconds, making $\rho = \frac{5000}{2500} = 2$. Note that since placement did not matter here we considered any 2 GPUs being used. Similarly ignoring placement, the bids for the other allocations are shown in Table 3.

We highlight a few more points about our example above. If the jobs that are chosen for the next phase do not match the median iteration time then the estimates are revised in the next round of the auction. For example, if the jobs that are chosen for the next round have iteration time 120, 100 then the above bid will be updated with $T_{sh(2)} = (120 \times 16) = 3200$[5] and $T_{sh(3)} = \frac{(120\times36)}{2} = 2160$. Further, we also see that the $job\_demand_{max} = 8$ means that the ρ value for 16 GPUs does not linearly decrease from that of 8 GPUs.

5 Implementation

We implement THEMIS on top of a recent release of Apache Hadoop YARN [1] (version 3.2.0) which includes, Submarine [2], a new framework for running ML training jobs atop YARN. We modify the Submarine client to support submitting a group of ML training jobs as required by hyper-parameter exploration apps. Once an app is submitted, it is managed by a Submarine Application Master (AM) and we make changes to the Submarine AM to implement the ML app scheduler (we use Hyperband [21]) and our AGENT.

To prepare accurate bids, we implement a profiler in the AM that parses TensorFlow logs, and tracks iteration times and loss values for all the jobs in an app. The allocation of a job changes over time and iteration times are used to accurately estimate the placement preference ($\mathcal{S}$) for different GPU placements. Loss values are used in our Hyperband implementation to determine early stopping. THEMIS's ARBITER is implemented as a separate module in YARN RM. We add gRPC-based interfaces between the AGENT and the ARBITER to enable offers, bids, and final winning allocations. Further, the ARBITER tracks GPU leases to offer reclaimed GPUs as a part of the offers.

All the jobs we use in our evaluation are TensorFlow programs with configurable hyper-parameters. To handle allocation changes at runtime, the programs checkpoint model parameters to HDFS every few iterations. After a change in allocation, they resume from the most recent checkpoint.

[5]Because the two jobs run on one GPU each, and the 120s-per-iteration job is the last to finish in the phase

6 Evaluation

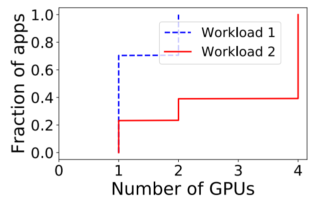
We evaluate THEMIS on a 64 GPU cluster and also use a event-driven simulator to model a larger 256 GPU cluster. We compare against other state-of-the-art ML schedulers. Our evaluation shows the following key highlights -

- THEMIS is better than other schemes on finish-time fairness while also offering better cluster efficiency (Figure 9-10-11-12).
- THEMIS's benefits compared to other schemes improve with increasing fraction of placement sensitive apps and increasing contention in the cluster, and these improvements hold even with errors – random and strategic – in finish-time fair metric estimations (Figure 14-18).
- THEMIS enables a trade-off between finish-time fairness in the long-term and placement efficiency in the short-term. Sensitivity analysis (Figure 19) using simulations show that $f = 0.8$ and a lease time of 10 minutes gives maximum fairness while also utilizing the cluster efficiently.

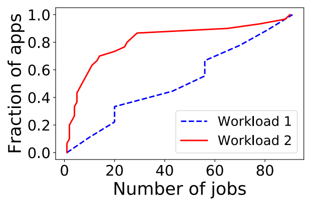
6.1 Experimental Setup

Testbed Setup. Our testbed is a 64 GPU, 20 machine cluster on Microsoft Azure [23]. We use NC-series instances. We have 8 NC12-series instances each with 2 Tesla K80 GPUs and 12 NC24-series instances each with 4 Tesla K80 GPUs.

Simulator. We develop an event-based simulator to evaluate THEMIS at large scale. The simulator assumes that estimates of the loss function curves for jobs are known ahead of time so as to predict the total number of iterations for the job. Unless stated otherwise, all simulations are done on a heterogeneous 256 GPU cluster. Our simulator assumes a 4-level hierarchical locality model for GPU placements. Individual GPUs fit onto



(a) CDF GPUs per job



(b) CDF jobs per app

Figure 8: Details of 2 workloads used for evaluation of THEMIS

	Model	Type	Dataset
10%	Inception-v3 [33]	CV	ImageNet [7]
	AlexNet [20]	CV	ImageNet
	ResNet50 [16]	CV	ImageNet
	VGG16 [32]	CV	ImageNet
	VGG19 [32]	CV	ImageNet
60%	Bi-Att-Flow [31]	NLP	SQuAD [28]
	LangModel [41]	NLP	PTB [22]
	GNMT [38]	NLP	WMT16 [37]
	Transformer [35]	NLP	WMT16
30%	WaveNet [25]	Speech	VCTK [40]
	DeepSpeech [15]	Speech	CommonVoice [6]

Table 4: Models used in our trace.

slots on *machines* occupying different cluster *racks*.[6]

Workload. We experiment with 2 different traces that have different workload characteristics in both the simulator and the testbed - **(i) Workload 1.** A publicly available trace of DNN training workloads at Microsoft [19,24]. We scale-down the trace, using a two week snapshot and focus on subset of jobs from the trace that correspond to hyper-parameter exploration jobs triggered by Hyperdrive. **(ii) Workload 2.** We use the app arrival times from Workload 1, generate jobs per app using the successive halving pattern characteristic of the Hyperband algorithm [21], and increase the number of tasks per job compared to Workload 1. The distribution of number of tasks per job and number of jobs per app for the two workloads is shown in Figure 8.

The traces comprise of models from three categories - computer vision (CV - 10%), natural language processing (NLP - 60%) and speech (Speech - 30%). We use the same mix of models for each category as outlined in Gandiva [39]. We summarize the models in Table 4.

Baselines. We compare THEMIS against four state-of-the-art ML schedulers - Gandiva [39], Tiresias [13], Optimus [27], SLAQ [42]; these represent the best possible baselines for maximizing efficiency, fairness, aggregate throughput, and aggregate model quality, respectively. We also compare against two scheduling disciplines - shortest remaining time first (SRTF) and shortest remaining service first (SRSF) [13]; these represent baselines for minimizing average job completion time (JCT) with efficiency as secondary concern and minimizing average JCT with fairness as secondary concern, respectively. We implement these baselines in our testbed as well as the simulator as described below:

Ideal Efficiency Baseline - Gandiva. Gandiva improves cluster utilization by packing jobs on as few machines as possible. In our implementation, Gandiva introspectively profiles ML job execution to infer placement preferences and migrates jobs to better meet these placement preferences. On any resource availability, all apps report their placement preferences and we allocate resources in a greedy highest preference first manner which has the effect of maximizing the average placement preference across apps. We do not model time-slicing and packing of GPUs as these system-level techniques can be integrated with THEMIS as well and would benefit Gandiva and THEMIS to equal extents.

Ideal Fairness Baseline - Tiresias. Tiresias defines a new service metric for ML jobs – the aggregate GPU-time allocated to each job – and allocates resources using the Least Attained Service (LAS) policy so that all jobs obtain equal service over time. In our implementation, on any resource availability, all apps report their service metric and we allocate the resource to apps that have the least GPU service.

Ideal Aggregate Throughput Baseline - Optimus. Optimus proposes a throughput scaling metric for ML jobs – the ratio of new job throughput to old job throughput with and without an additional GPU allocation. On any resource availability, all apps report their throughput scaling and we allocate resources in order of highest throughput scaling metric first.

Ideal Aggregate Model Quality - SLAQ. SLAQ proposes a greedy scheme for improving aggregate model quality across all jobs. In our implementation, on any resource availability event, all apps report the decrease in loss value with allocations from the available resources and we allocate these resources in a greedy highest loss first manner.

Ideal Average App Completion Time - SRTF, SRSF. For SRTF, on any resource availability, all apps report their remaining time with allocations from the available resource and we allocate these resources using SRTF policy. Efficiency is a secondary concern with SRTF as better packing of GPUs leads to shorter remaining times.

SRSF is a service-based metric and approximates gittins index policy from Tiresias. In our implementation, we assume accurate knowledge of remaining service and all apps report their remaining service and we allocate one GPU at a time using SRSF policy. Fairness is a secondary concern as shorter service apps are preferred first as longer apps are more amenable to make up for lost progress due to short-term unfair allocations.

Metrics. We use a variety of metrics to evaluate THEMIS.

(i) Finish-time fairness: We evaluate the fairness of schemes by looking at the finish-time fair metric (ρ) distribution and the maximum value across apps. A tighter distribution and a lower value of maximum value of ρ across apps

[6]The heterogeneous cluster consists of 16 8-GPU machines (4 slots and 2 GPUs per slot), 6 4-GPU machines (4 slots and 1 GPU per slot), and 16 1-GPU machines

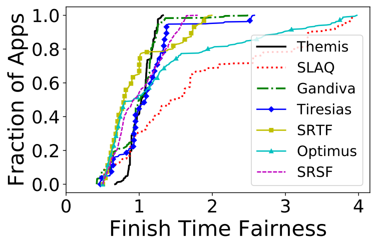


Figure 9: [TESTBED] Comparison of finish-time fairness across schedulers with Workload 1

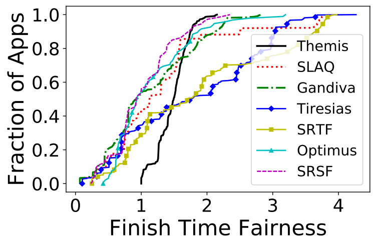


Figure 10: [TESTBED] Comparison of finish-time fairness across schedulers with Workload 2

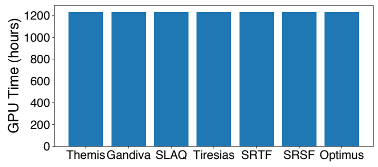


Figure 11: [TESTBED] Comparison of total GPU times across schemes with Workload 1. Lower GPU time indicates better utilization of the GPU cluster

indicate higher fairness. **(ii) GPU Time:** We use *GPU Time* as a measure of how efficiently the cluster is utilized. For two scheduling schemes S_1 and S_2 that have GPU times G_1 and G_2 for executing the same amount of work, S_1 utilizes the cluster more efficiently than S_2 if $G_1 < G_2$. **(iii) Placement Score:** We give each allocation a placement score (≤ 1). This is inversely proportional to slowdown, $\mathcal{S}$, that app experiences due to this allocation. The slowdown is dependent on the ML app properties and the network interconnects between the allocated GPUs. A placement score of 1.0 is desirable for as many apps as possible.

6.2 Macrobenchmarks

In our testbed, we evaluate THEMIS against all baselines on all the workloads. We set the fairness knob value f as 0.8 and lease as 10 minutes, which is informed by our sensitivity analysis results in Section 6.4.

Figure 9-10 shows the distribution of finish-time fairness metric, ρ, across apps for THEMIS and all the baselines. We see that THEMIS has a narrower distribution for the ρ values which means that THEMIS comes closest to giving all jobs an equal sharing incentive. Also, THEMIS gives 2.2X to 3.25X better (smaller) maximum ρ values compared to all baselines.

Figure 11-12 shows a comparison of the efficiency in terms of the aggregate GPU time to execute the complete workload. Workload 1 has similar efficiency across THEMIS and the baselines as all jobs are either 1 or 2 GPU jobs and almost all allocations, irrespective of the scheme, end up as efficient. With workload 2, THEMIS betters Gandiva by ~4.8% and outperforms SLAQ by ~250%. THEMIS is better because global visibility of app placement preferences due to the auction abstraction enables globally optimal decisions. Gandiva in contrast takes greedy locally optimal packing decisions.

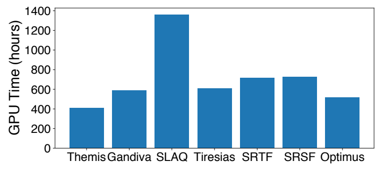


Figure 12: [TESTBED] Comparison of total GPU times across schemes with Workload 2. Lower GPU time indicates better utilization of the GPU cluster

Job Type	GPU Time	# GPUs	ρ_{THEMIS}	$\rho_{Tiresias}$
Long Job	~580 mins	4	~1	~0.9
Short Job	~83 mins	2	~1.2	~1.9

Table 5: [TESTBED] Details of 2 jobs to understand the benefits of THEMIS

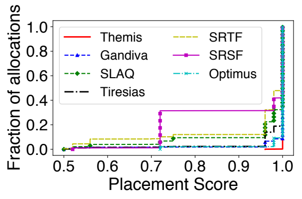


Figure 13: [TESTBED] CDF of placement scores across schemes

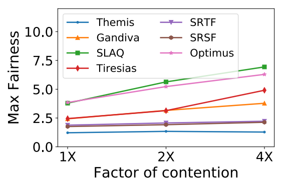


Figure 14: [TESTBED] Impact of contention on finish-time fairness

6.2.1 Sources of Improvement

In this section, we deep-dive into the reasons behind the wins in fairness and cluster efficiency in THEMIS.

Table 5 compares the finish-time fair metric value for a pair of short- and long-lived apps from our testbed run for THEMIS and Tiresias. THEMIS offers better sharing incentive for both the short and long apps. THEMIS induces altruistic behavior in long apps. We attribute this to our choice of ρ metric. With less than ideal allocations, even though long apps see an increase in T_{sh}, their ρ values do not increase drastically because of a higher T_{id} value in the denominator. Whereas, shorter apps see a much more drastic degradation, and our round-by-round filtering of farthest-from-finish-time fairness apps causes shorter apps to participate in auctions more often. Tiresias offers poor sharing incentive for short apps as it treats short- and long-apps as the same. This only worsens the sharing incentive for short apps.

Figure 13 shows the distribution of placement scores for all the schedulers. THEMIS gives the best placement scores (closer to 1.0 is better) in workload 2, with Gandiva and Optimus coming closest. Workload 1 has jobs with very low GPU demand and almost all allocations have a placement score of 1 irrespective of the scheme. Other schemes are poor as they do not account for placement preferences. Gandiva does greedy local packing and Optimus does greedy throughput scaling and are not as efficient because they are not globally optimal.

6.2.2 Effect of Contention

In this section, we analyze the effect of contention on finish-time fairness. We decrease the size of the cluster to half and quarter the original size to induce a contention of $2X$ and $4X$ respectively. Figure 14 shows the change in max value of ρ as the contention changes with workload 1. THEMIS is the only scheme that maintains *sharing incentive* even in high contention scenarios. SRSF comes close as it preferably allocates resources to shorter service apps. This behavior is similar to that in THEMIS. THEMIS induces altruistic shedding of resources by longer apps (Section 6.2.1), giving shorter apps a preference in allocations during higher contention.

6.2.3 Systems Overheads

From our profiling of the experiments above, we find that each AGENT spends 29 (334) milliseconds to compute bids at the median (95-%). The 95 percentile is high because enumeration of possible bids needs to traverse a larger search space when the number of resources up for auction is high.

The ARBITER uses Gurobi [14] to compute partial allocation of resources to apps based on bids. This computation takes 354 (1398) milliseconds at the median (95-%ile). The high tail is once again observed when both the number of offered resources and the number of apps bidding are high. However, the time is small relative to lease time. The network overhead for communication between the ARBITER and individual apps is negligible since we use the existing mechanisms used by Apache YARN.

Upon receiving new resource allocations, the AGENT changes (adds/removes) the number of GPU containers available to its app. This change takes about 35 (50) seconds at the median (95-%ile), i.e., an overhead of 0.2% (2%) of the app duration at the median (95-%ile). Prior to relinquishing control over its resources, each application must checkpoint its set of parameters. We find that that this is model dependent but takes about 5-10 seconds on an average and is driven largely by the overhead of check-pointing to HDFS.

6.3 Microbenchmarks

Placement Preferences: We analyze the impact on finish-time fairness and cluster efficiency as the fraction of network-intensive apps in our workload increases. We synthetically construct 6 workloads and vary the percentage of network-intensive apps in these workloads from 0%-100%.

From Figure 15, we notice that *sharing incentive* degrades most when there is a heterogeneous mix of compute and network intensive apps (at 40% and 60%). THEMIS has a max ρ value closest to 1 across all scenarios and is the only scheme to ensure sharing incentive. When the workload consists solely of network-intensive apps, THEMIS performs ~1.24 to 1.77X better than existing baselines on max fairness.

Figure 16 captures the impact on cluster efficiency. With only compute-intensive apps, all scheduling schemes utilize the cluster equally efficiently. As the percentage of network intensive apps increases, THEMIS has lower GPU times to execute the same workload. This means that THEMIS utilizes the cluster more efficiently than other schemes. In the workload with 100% network-intensive apps, THEMIS performs ~8.1% better than Gandiva (state-of-the-art for cluster efficiency).

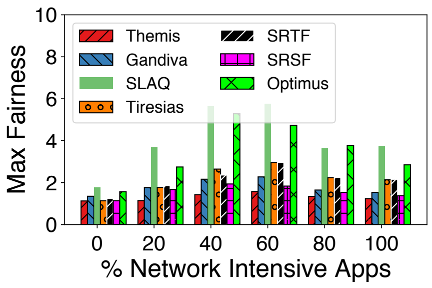


Figure 15: [SIMULATOR] Impact of placement preferences for varying mix of compute- and network-intensive apps on max ρ

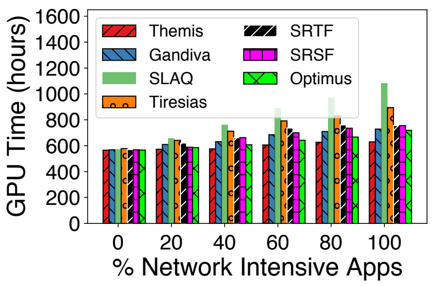


Figure 16: [SIMULATOR] Impact of placement preferences for varying mix of compute- and network-intensive apps on GPU Time

Error Analysis: Here, we evaluate the ability of THEMIS to handle errors in estimation of number of iterations and the slowdown ($\mathcal{S}$). For this experiment, we assume that all apps are equally susceptible to making errors in estimation. The percentage error is sampled at random from [-X, X] range for each app. Figure 17 shows the changes in max finish-time fairness as we vary X. Even with $X = 20\%$, the change in max finish-time fairness is just 10.76% and is not significant.

Truth-Telling: To evaluate strategy-proofness, we use simulations. We use a cluster of 64 GPUs with 8 identical apps with equivalent placement preferences. The cluster has a single 8 GPU machine and the others are all 2 GPU machines. The most preferred allocation in this cluster is the 8 GPU machine. We assume that there is a single strategically lying app and 7 truthful apps. In every round of auction it participates in, the lying app over-reports the slowdown with staggered machine placement or under-reports the slowdown with dense machine placement by $X\%$. Such a strategy would ensure higher likelihood of winning the 8 GPU machine. We vary the value of X in the range $[0, 100]$ and analyze the lying app's completion time and the average app completion time of the truthful apps in Figure 18. We see that at first the lying app does not experience any decrease in its own app completion time. On the other hand, we see that the truthful apps do better on their average app completion time. This is because the hidden payment from the partial allocation mechanism in each round of the auction for the lying app remains the same while

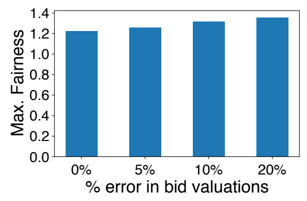


Figure 17: [SIMULATOR] Impact of error in bid values on max fairness

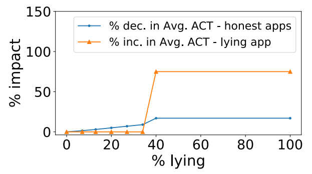
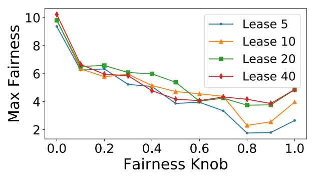
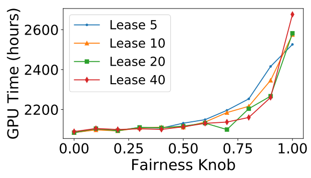


Figure 18: [SIMULATOR] Strategic lying is detrimental



(a) Impact on Max Fairness



(b) Impact on GPU Time

Figure 19: [SIMULATOR] Sensitivity of fairness knob and lease time.

the payment from the rest of the apps keeps decreasing. We also observe that there is a sudden tipping point at $X > 34\%$. At this point, there is a sudden increase in the hidden payment for the lying app and it loses a big chunk of resources to other apps. In essence, THEMIS incentivizes truth-telling.

6.4 Sensitivity Analysis

We use simulations to study THEMIS's sensitivity to fairness knob f and the lease time. Figure 19 (a) shows the impact on max ρ as we vary the fairness knob f. We observe that filtering $(1-f)$ fraction of apps helps with ensuring better *sharing incentive*. As f increases from 0 to 0.8, we observe that fairness improves. Beyond $f = 0.8$, max fairness worsens by around a factor of $1.5X$. We see that the quality of sharing incentive, captured by max ρ, degrades at $f = 1$ because we observe that only a single app with highest ρ value participates in the auction. This app is forced sub-optimal allocations because of poor placement of available resources with respect to the already allocated resources in this app. We also observe that smaller lease times promote better fairness since frequently filtering apps reduces the time that queued apps wait for an allocation.

Figure 19 (b) shows the impact on the efficiency of cluster usage as we vary the fairness knob f. We observe that the efficiency decreases as the value of f increases. This is because the number of apps that can bid for an offer reduces as we increase f leading to fewer opportunities for the ARBITER to pack jobs efficiently. Lower lease values mean than models need to be check-pointed more often (GPUs are released on lease expiry) and hence higher lease values are more efficient.

Thus we choose $f = 0.8$ and *lease* $= 10$ minutes.

7 Related Work

Cluster scheduling for ML workloads has been targeted by a number of recent works including SLAQ [42], Gandiva [39], Tiresias [13] and Optimus [27]. These systems target different objectives and we compare against them in Section 6.

We build on rich literature on cluster scheduling disciplines [8, 10–12] and two level schedulers [17, 30, 36]. While those disciplines/schedulers don't apply to our problem, we build upon some of their ideas, e.g., resource offers in [17]. Sharing incentive was outlined by DRF [8], but we focus on long term fairness with our finish-time metric. Tetris [10] proposes resource-aware packing with an option to trade-off for fairness using multi-dimensional bin-packing as the mechanism for achieving that. In THEMIS, we instead focus on fairness with an option to trade-off for placement-aware packing, and use auctions as our mechanism.

Some earlier schemes [11, 12] also attempted to emulate the long term effects of fair allocation. Around occasional barriers, unused resources are re-allocated across jobs. THEMIS differs in many respects: First, earlier systems focus on batch analytics. Second, earlier schemes rely on instantaneous resource-fairness (akin to DRF), which has issues with placement-preference unawareness and not accounting for long tasks. Third, in the ML context there are no occasional barriers. While barriers do arise due to synchronization of parameters in ML jobs, they happen at *every* iteration. Also, resources unilaterally given up by a job may not be usable by another job due to placement preferences.

8 Conclusion

In this paper we presented THEMIS, a fair scheduling framework for ML training workloads. We showed how existing fair allocation schemes are insufficient to handle long-running tasks and placement preferences of ML workloads. To address these challenges we proposed a new long term fairness objective in finish-time fairness. We then presented a two-level semi-optimistic scheduling architecture where ML apps can bid on resources offered in an auction. Our experiments show that THEMIS can improve fairness *and* efficiency compared to state of the art schedulers.

Acknowledgements. We are indebted to Varun Batra and Surya Teja Chavali for early discussions and helping with cluster management. We thank the Azure University Grant for their generous support in providing us the GPU resources used for experiments in this paper. We also thank Jim Jernigan for sharing his insights on running large GPU clusters at Microsoft. Finally, we thank the reviewers and our shepherd Manya Ghobadi. This work is supported by the National Science Foundation (grants CNS-1838733, CNS-1763810, CNS-1563095, CNS-1617773, and CCF-1617505). Shivaram Venkataraman is also supported by a Facebook faculty research award and support for this research was also provided by the Office of the Vice Chancellor for Research and Graduate Education at the University of Wisconsin, Madison with funding from the Wisconsin Alumni Research Foundation. Aditya Akella is also supported by a Google faculty research award, a Facebook faculty research award, and H. I. Romnes Faculty Fellowship.

References

[1] Apache Hadoop NextGen MapReduce (YARN). Retrieved 9/24/2013, URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2013.

[2] Apache Hadoop Submarine. https://hadoop.apache.org/submarine/, 2019.

[3] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. D. Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1), 2015.

[4] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[5] R. Cole, V. Gkatzelis, and G. Goel. Mechanism design for fair division: allocating divisible items without payments. In *Proceedings of the fourteenth ACM conference on Electronic commerce*, pages 251–268. ACM, 2013.

[6] Common Voice Dataset. https://voice.mozilla.org/.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[8] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[9] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *KDD*, 2017.

[10] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2015.

[11] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 65–80, 2016.

[12] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, 2016.

[13] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 485–500, 2019.

[14] Gurobi Optimization. http://www.gurobi.com/.

[15] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[18] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[19] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX ATC*, 2019.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[21] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[22] M. Marcus, B. Santorini, and M. A. Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993.

[23] Microsoft Azure. https://azure.microsoft.com/en-us/.

[24] Microsoft Philly Trace. https://github.com/msr-fiddle/philly-traces, 2019.

[25] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[26] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *SOSP*, pages 69–84, 2013.

[27] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, page 3. ACM, 2018.

[28] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[29] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca. Hyperdrive: Exploring hyperparameters with pop scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 1–13. ACM, 2017.

[30] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Eurosys*, 2013.

[31] M. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[32] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[33] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[34] H. R. Varian. Equity, envy, and efficiency. *Journal of Economic Theory*, 9(1):63 – 91, 1974.

[35] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[36] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Eurosys*, 2015.

[37] WMT16 Dataset. http://www.statmt.org/wmt16/.

[38] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[39] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[40] J. Yamagishi. English multi-speaker corpus for cstr voice cloning toolkit. *URL http://homepages. inf. ed. ac. uk/jyamagis/page3/page58/page58. html*, 2012.

[41] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[42] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 390–404. ACM, 2017.

A Appendix

PROOF OF THEOREM 3.1. Examples in Figure 5 and Section 3.1.2 shows that DRF violates SI, EF, and PE. Same examples hold true for LAS policy in Tiresias. The service metric i.e. the GPU in Instance 1 and Instance 2 is the same for A1 and A2 in terms of LAS and is deemed a fair allocation over time. However, Instance 1 violates SI as A1 (VGG16) and A2 (VGG16) would prefer there own independent GPUs and Instance 2 violates EF and PE as A2 (VGG16) prefers the allocation of A1 (Inception-v3) and PE as the optimal allocation after taking into account placement preferences would interchange the allocation of A1 and A2. □

PROOF OF THEOREM 3.2. We first show that the valuation function, $\rho(.)$, for the case of ML jobs is homogeneous. This means that $\rho(.)$ has the following property: $\rho(m * \overrightarrow{G}) = m * \rho\overrightarrow{G}$.

Consider a job with GPUs spread across a set of some M machines. If we keep this set of machines the same, and increase the number of GPUs allocated on these same set of machines by a certain factor then the shared running time (T_{sh}) of this job decreases proportionally by the same factor. This is so because the slowdown, $\mathcal{S}$ remains the same. Slowdown is determined by the slowest network interconnect between the machines. The increased allocation does not change the set of machines M. The independent running time (T_{id}) remains the same. This means that ρ also proportionally changes by the same factor.

Given, homogeneous valuation functions, the PA mechanism guarantees SP, PE and EF [5]. However, PA violates SI due to the presence of hidden payments. This also make PA not work-conserving. □

PROOF OF THEOREM 3.3. With multi-round auctions we ensure truth-telling of ρ estimates in the visibility phase. This is done by the AGENT by using the cached $\rho(.)$ estimates from the last auction the app participated in. In case an app gets leftover allocations from the leftover allocation mechanism, the AGENT updates the ρ estimate again by using the cached $\rho(.)$ table. In this way we guarantee SP with multi-round auctions.

As we saw in Theorem 3.2, an auction ensures PE and EF. In each round, we allocate all available resources using auctions. This ensures end-to-end PE and EF.

For maximizing sharing incentive, we always take a fraction $1 - f$ of apps in each round. A wise choice of f ensures that we filter in all the apps with $\rho > 1$ that have poor sharing incentive. We only auction the resources to such apps which maximizes sharing incentive. □

Fine-Grained Replicated State Machines for a Cluster Storage System

Ming Liu[*] *Arvind Krishnamurthy*[*] *Harsha V. Madhyastha*[†] *Rishi Bhardwaj*[‡] *Karan Gupta*[‡]

Chinmay Kamat[‡] *Huapeng Yuan*[‡] *Aditya Jaltade*[‡] *Roger Liao*[‡] *Pavan Konka*[‡] *Anoop Jawahar*[‡]

Abstract

We describe the design and implementation of a consistent and fault-tolerant metadata index for a scalable block storage system. The block storage system supports the virtualized execution of legacy applications inside enterprise clusters by automatically distributing the stored blocks across the cluster's storage resources. To support the availability and scalability needs of the block storage system, we develop a distributed index that provides a replicated and consistent key-value storage abstraction.

The key idea underlying our design is the use of fine-grained replicated state machines, wherein every key-value pair in the index is treated as a separate replicated state machine. This approach has many advantages over a traditional coarse-grained approach that represents an entire shard of data as a state machine: it enables effective use of multiple storage devices and cores, it is more robust to both short- and long-term skews in key access rates, and it can tolerate variations in key-value access latencies. The use of fine-grained replicated state machines, however, raises new challenges, which we address by co-designing the consensus protocol with the data store and streamlining the operation of the per-key replicated state machines. We demonstrate that fine-grained replicated state machines can provide significant performance benefits, characterize the performance of the system in the wild, and report on our experiences in building and deploying the system.

1 Introduction

Enterprise clusters often rely on the abstraction of a block storage volume to support the virtualized execution of applications. Block storage volumes appear as local disks to virtual machines running legacy applications, even as the storage service distributes volume data across the cluster. The storage system provides ubiquitous access to volumes from any node in the cluster and ensures durability and availability through replication.

Our work is in the context of a commercial enterprise cluster product built by Nutanix, a software company that specializes in building private clouds for enterprises. VMs deployed in these clusters rely on a cluster block storage system, called Stargate. As with other block storage systems [8,10,27,29,31], Stargate provides a virtual disk abstraction on which applications/VMs can instantiate any file system. However, unlike most other block storage systems, Stargate co-locates both computing and storage on the same set of cluster nodes. This approach provides cost, latency, and scalability benefits: it avoids needing to provision separate resources for computing and storage, it allows for local access to storage, and it lets both storage and compute scale with the cluster size.

A key component of such a system is the metadata index, which maps the logical blocks associated with a virtual disk to its actual physical locations. Just like the overall system, this mapping layer should provide high performance and strong consistency guarantees in the presence of failures. These requirements suggest a design with the following elements: (a) achieve high throughput and scalability by distributing the index as key-value pairs and utilizing all the cluster nodes, (b) ensure availability and consistency by replicating key-value pairs and using a consensus algorithm, such as Paxos [16] or Viewstamped Replication [25], to implement replicated state machines (RSMs), and (c) ensure durability of a node's shard of key-value state by employing a node-level durable data structure such as the log-structured merge tree (LSM).

This traditional approach to building a distributed index has drawbacks in our specific context where: (a) all operations, including metadata operations, have to be made durable before they are acknowledged, (b) there is significant variation in operation execution latency, and (c) the distributed index service has to share compute and storage with the rest of Stargate and application VMs. In particular, the use of a per-shard consensus operation log, which records the order of issued commands, introduces inefficiencies, such as short- and long-term load imbalances on storage devices, sub-optimal batching of storage operations, and head-of-line blocking caused by more expensive operations.

To address these issues, we develop a design that uses *fine-grained replicated state machine (fRSMs)*, where each key-value pair is represented as a separate RSM and can operate independently. This approach allows for flexible and dynamic scheduling of operations on the metadata service and enables effective use of the storage and compute resources. To efficiently realize this approach, we use a combination of techniques to streamline the state associated with the object radically. In particular, our approach uses no operation logs and maintains only a small amount of consensus state along with the perceived value of a key. We also address performance and consistency issues by co-designing the consensus protocol and the local node storage, providing strong guarantees on operation orderings, and optimizing failure recovery by enhancing the LSM data structure to handle the typical failure scenarios efficiently. It is worth noting that our innovation is not in the consensus protocol (as we merely borrow elements from Paxos and Viewstamped Replication), but in exploring

[*]University of Washington
[†]University of Michigan
[‡]Nutanix

an extreme operating point that is appropriate for balancing load across storage and compute resources in a managed environment with low downtimes.

We present experimental evaluations of our implementation both in a controlled testbed as well as in production deployments. Compared with traditional coarse-grained RSMs, fRSMs achieve 5.6× and 2.3× higher throughput for skewed and uniform scenarios in controlled testbeds. The resulting implementation is part of a commercial storage product that we have deployed on thousands of clusters over the past eight years. To date, we have not had a data loss event at any of these deployed production sites. We have also been able to leverage the metadata store for other applications such as write-ahead logs and distributed hypervisor management.

2 Motivation

We begin with a description of our setting and our goals. We then describe a baseline approach and discuss its shortcomings that motivate our work.

2.1 Metadata Storage Overview

Setting. Our work targets clusters that are typically used by enterprises as private clouds to perform on-premise computing. Customers instantiate virtual machines (VMs) that run legacy applications. The cluster management software then determines which node to run each VM on, migrating them as necessary to deal with faults and load imbalances.

Our Stargate storage system provides a virtual disk abstraction to these VMs. VMs perform reads and writes on the virtual disk blocks, and Stargate translates them to the appropriate accesses on physical disks that store the corresponding blocks. Stargate stores the blocks corresponding to virtual disks on any one of the cluster nodes on which user VMs are executed, thus realizing a *hyper-converged cluster infrastructure* that co-locates compute and storage. An alternate approach would be to use a separate cluster of storage nodes (as is the case with solutions such as SAN) and provide the virtual disk abstraction over the network. Nutanix employs co-location as it reduces infrastructure costs and allows the storage system to flexibly migrate data blocks accessed by a VM to the node on which the VM is currently hosted, thereby providing low latency access and lowering network traffic.

Metadata storage. In this paper, we focus on how Stargate stores the metadata index that maps virtual disk blocks to physical locations across the cluster. One can implement the virtual disk abstraction by maintaining a map for each virtual disk (vDisk) that tracks the physical disk location for every block in that vDisk. Our design, outlined below, introduces additional levels of indirection to support features such as deduplication, cloning and snapshotting. It also separates *physical maps* from *logical maps* to allow for decoupled updates to these maps.

A virtual disk is a sequence of *extents*, each of which is identified by an *ExtentID*. An extent can be shared across virtual disks either because of the deduplication of disk blocks

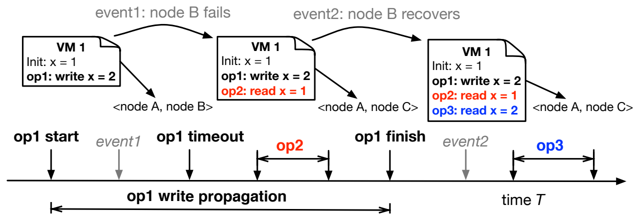


Figure 1: Example timeline that satisfies linearizability but complicates reasoning about failures. The notation <node A, node B> means that the VM is on node A and the leader of the replica group maintaining key *X* is on node B. The value of key *x* is 1 at the start of the timeline. A VM, initially running on node A, issues a write to *x*, partially performs it on node B, and suffers a timeout due to B's failure. After another node C becomes the leader, the VM reads 1 from *x* and expects to continue to see *x* set to 1, barring new writes issued subsequently. If the old leader were to recover, it could propagate its updated copy of *x* and interfere with the VM's logic.

or snapshotting/cloning of virtual disks. Extents are grouped into units called extent groups, each of which has an associated *ExtentGroupID*, and each extent group is stored as a contiguous unit on a storage device. Given this structure, the storage system uses the *vDisk Block Map* to map portions of a vDisk to ExtentIDs, the *ExtentID Map* to map extents to ExtentGroupIDs, and the *ExtentGroupID Map* to map ExtentGroupIDs to physical disk locations. These maps are shared between VMs and the cluster storage management system, which might move, compress, deduplicate, and garbage-collect storage blocks. All accesses to a given vDisk are serialized through a vDisk controller hosted on one of the cluster nodes. Stargate migrates vDisk controllers and VMs upon node failures.

Goals. In determining how to store Stargate's metadata index, apart from maximizing availability and efficiency, we have the following goals:

- ***Durability:*** To minimize the probability of data loss, any update to the metadata must be committed to stable storage on multiple nodes in the cluster before Stargate acknowledges the write as complete to the client. Note that our system should maintain consistent metadata even when the entire cluster comes down (e.g., due to a correlated failure).
- ***Consistency:*** Operations on the metadata index should be linearizable, i.e., all updates to a block's metadata should be totally ordered, and any read should return the last completed write. This guarantee provides strong consistency semantics to client VMs and various background services that operate on the metadata.
- ***Reasoning about failures:*** Under linearizability, even if a read issued after a failure does not reflect a write issued before the failure, this does not mean that the write failed; the update could have been arbitrarily delayed and might get applied later, causing subsequent reads to observe the updated value (see Figure 1). The system should provide stronger guarantees to client VMs so that they can reason about operation failures. In particular, any subsequent read

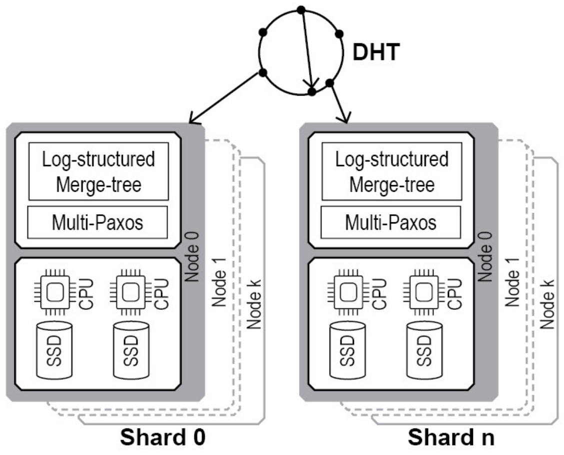


Figure 2: Baseline system architecture representing a coarse-grained replicated state machine built using LSM and Paxos.

of the metadata after an operation timeout must confirm whether the prior operation succeeded or not, and successive reads of a piece of metadata should return the same value as long as there are no concurrent updates initiated by other agents in the system.

2.2 Baseline Design

Let us now consider a baseline approach for realizing the above-mentioned goals. This baseline takes the traditional approach of (a) sharding the metadata index across multiple nodes and multiple cores or SSDs on a given node, (b) using a consensus protocol for ordering operations on any given shard, and (c) executing operations on a durable data structure such as a log-structured merge tree.

In the baseline, all nodes in the cluster participate in implementing a distributed key-value store. We partition keys into shards, use consistent hashing to map shards to replica sets, and consider a leader-based consensus protocol wherein each node serves as the leader for one or more shards in the system. Leader-less designs (such as EPaxos [23]) can lower the communication costs as they eliminate the coordination overheads for the leader, but provide limited benefits in our setting. First, when storage and compute are co-located, there is limited value in moving communication costs from the leader to a client that is sharing network resources with a different server in the cluster. Second, as we will demonstrate later, storage and compute resources are bigger bottlenecks in our setting than the network. Due to our design choice of co-locating compute and storage, the metadata service shares resources with client VMs, which have a higher priority.

We consider a layered design wherein the lower layer corresponds to a consensus protocol, and the upper layer corresponds to a state machine implementing a durable data structure such as a log-structured merge tree. The timeline for processing a request proceeds as follows.

Consensus layer processing. For every shard, one of the replicas of the shard becomes the leader by sending "prepare" messages to a quorum of replicas. When the leader receives a mutating command such as a write, it sequences and propagates this command to all replicas (including itself) using a consensus protocol such as Paxos [16], Viewstamped Replication [25], or Raft [26]. Each shard that a node is assigned to is associated with a specific core and a specific SSD on that node; the core is responsible for sequencing updates to the shard, and the corresponding operation log is stored on the SSD. The system maximizes efficiency by committing commands to the SSD in batches, with every node batching updates destined to one of its SSDs until the prior write to that SSD is complete. Once a batched write is completed, all operations in that batch are considered "accepted". After the leader receives a quorum number of accepts for a command, it can then execute the command locally and send "learn" messages to all followers, indicating that the command has been "chosen." The chosen status does not have to be recorded in stable storage as it can be recreated upon failures. A centralized approach with primary-backup replication [3] can eliminate the use of a consensus protocol and simplify the system design. Such a design, however, limits both the operational scale and performance, and wouldn't satisfy the system requirements that we had outlined above.

LSM layer processing. At every node, the LSM layer processes all chosen commands in the order determined by the consensus layer. LSM processing is streamlined to include just the in-memory Memtable and the stable SSTables. In particular, this is a slightly customized version of a traditional LSM implementation as the commit log, which is available from the consensus layer, can be eliminated from the LSM code. The Memtable access and compaction operations need to be synchronized with other concurrent operations to support multi-core operations. The leader acknowledges a command as complete to the client after a quorum of nodes has recorded the command, and the leader has executed the command in its chosen order because the success of some commands (e.g., compare-and-swap) can be determined only when they are executed after all previously accepted commands have been applied. Leases enable the leader to serve reads on the LSM without any communication with other nodes. However, the leader must synchronize every read on a key with ongoing updates to the same key.

Ordering guarantees. RSMs built using consensus protocols provide linearizability. Further, an RSM can guarantee in-order execution of operations issued by a client. This helps the client reason about the execution status of its operations that have timed out – if the result of a later operation implies that an earlier operation has not been performed, the client can not only deduce that the prior operation has not yet completed but also get the guarantee that the service will never perform the operation. This guarantee can be provided even after RSM reconfigurations. Upon leadership and view changes, protocols such as Viewstamped Replication ensure that operations partially performed in a previous view are not completed in sub-

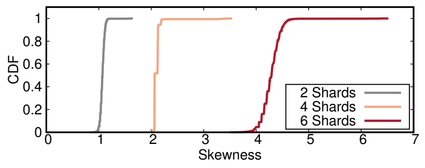


Figure 3: CDF of access skewness with 2/4/6 data shards. ***skewness*** **at any instant is defined as the ratio of the maximum to the average of outstanding IOs per shard.**

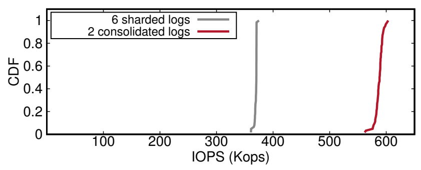


Figure 4: CDF of aggregate SSD throughput when 6 commit logs (3 per SSD) are used compared to when 2 commit logs (one per SSD) are used.

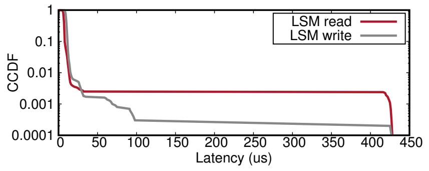


Figure 5: CCDF of LSM random 4KB read/write latencies. The 99.9th percentile latency for LSM reads/writes is 57.1×/48.4× the respective averages.

sequent views. These guarantees provide clients with some capability to infer the completion status of their operations.

2.3 Performance Implications of Baseline Design

This baseline design, however, results in several sources of inefficiency. We quantify them with micro-benchmarks using the same computing setup as our evaluations (see Section 4.1).

- ***Load imbalance due to skew:*** The skew in load across shards can lead to an imbalance across SSDs and CPU cores. For instance, differences in popularity across keys can result in long-term skew, whereas random Poisson arrival of requests can cause short-term skew. Figure 3 quantifies the skews across shards for random Poisson arrival.
- ***Sub-optimal batching:*** If there are n nodes in a replica set, each with m SSDs, the number of shards into which commands would be accumulated would be the least common multiple of m and n. (This ensures that the assignment of shard storage to SSDs and the assignment of shard leadership to nodes are statically balanced.) Batching updates independently on each of these shards can result in less than optimal latency amortization. Figure 4 shows that batching across multiple data shards can achieve 1.6× higher bandwidth than a traditional per-shard log design.
- ***High tail latency:*** Tail latency or even average latency of operations could be high due to multiple reasons. First, since the RSM abstraction requires that all replicas execute all updates in the same order, if one of the replicas for a shard is missing a command in its commit log, subsequent operations on that shard will block until this replica catches up. Second, since LSM operations vary in terms of their execution costs (shown in Figure 5), a heavyweight operation can delay the execution of lightweight operations even if processor cores are available to execute the operations.

Sub-dividing the shards into even smaller shards would mitigate the load imbalance issue. However, it suffers from three drawbacks. First, it doesn't address the request head-of-line blocking issue. Requests still have to commit and execute in sequence, as specified in the log order. Second, it further reduces batching efficiency for storage devices. Third, it doesn't provide the benefit of fast node recovery, as a recovering node cannot immediately participate in the protocol. As a result, we instead adopt a shard-less design to overcome all of these issues, as we describe next.

3 System Design

We now present the design of Stargate's metadata storage system, which provides the desired efficiency, availability, durability, and consistency properties. We use the same high-level approach as the baseline: consistent hashing to distribute metadata across replica sets, log-structured merge trees to store and access large, durable datasets, and a consensus protocol to ensure consistency of operations on replicated data.

Our approach differs in one fundamental aspect: it uses fine-grained replicated state machines (fRSMs), wherein each replicated key is modeled as a separate RSM. This approach provides the flexibility needed to effectively manage multiple storage devices and CPU cores on a server, reduces load imbalances, and enables flexible scheduling of key-value operations. However, the use of fine-grained state machines raises both performance and consistency issues, and we address them by carefully co-designing the consensus protocol, the data store, and the client stubs that interact with the storage layer.

3.1 Overview and Design Roadmap

Replicating every key-value pair as a separate RSM, though conceptually straightforward, could impose significant overheads because RSMs are rather heavyweight. For example, a typical RSM contains an operation log and consensus state for each entry in the operation log. The operation log is used to catch up replicas that are lagging behind and/or have missing entries; each operation in the log has to be propagated to laggards to get their state up-to-date.

Lightweight RSMs. Fortunately, the RSM state can be vastly streamlined for simple state machines, such as the key-value objects we use in our system.

- For normal read/write and synchronizing operations such as compare-and-swap, the next state of a key-value pair is a function of its current state and any argument that is provided along with the operator. For such operations, one can eliminate the need for an operation log; it suffices to maintain just the last mutating operation that has been performed on the object and any in-progress operations being performed on the object. We use an API that is simple and yet sufficiently powerful to support the metadata operations of a cluster storage system. (See Section 3.2.)
- The consensus state for operations on a key (e.g., promised and accepted proposal numbers) is stored along with the

key-value state in the LSM as opposed to requiring a separate data structure for maintaining this information. (See Section 3.3.1.)

- A consensus protocol typically stores accepted but not yet committed values along with its committed state and commits an accepted value when consensus has been reached. Instead, our system speculatively executes the operations, stores the resulting value in a node's LSM, and relies on the consensus protocol to update this to the consensus value for the key in the case of conflicts. This further reduces the RSM state associated with each key. It also eliminates the need for explicit learn messages.[1] (See Section 3.3.2.)
- Similar to the Vertical Paxos approach [18], leader election is performed on a per key-range granularity using a separate service (e.g., Zookeeper [13] in our case).

Enabled optimizations. We co-design the consensus protocol and the LSM layer implementing the key-value store to realize per-key RSMs.[2] This enables many optimizations.

- *Consolidated LSM:* All the key-values replicated on a given node can be stored in a single LSM tree as opposed to the canonical sharded implementation that would require a separate LSM tree for each shard. The commit log of the unified LSM tree can be striped across the different storage devices, thus leading to more effective batching of I/O requests to the commit log.
- *Load balancing:* Per-key RSMs enable flexible and late binding of operation processing to CPU cores; a key-value operation can be processed on any core (as long as there is per-key in-memory synchronization to deal with concurrency) and durable updates can be performed on any SSD, leading to more balanced utilization of cores and SSDs.
- *Minimizing stalls:* By requiring ordering of operations only per-key, rather than per-shard, we can eliminate head-of-line blocking. Message loss and high-latency LSM operations do not impact the performance of ongoing operations on other keys, thus improving the tail latency of operations.
- *Low-overhead replication:* Each operation can be applied to just a quorum of replicas (e.g., two nodes in a replica set of three), thus increasing the overall throughput that the system can support. With coarse-grained RSMs, this optimization would result in a period of unavailability whenever a node fails, because new operations on a shard can only be served after stale nodes catch up on all previous operations on the shard. With fRSMs, lagging nodes can be updated on a per-key basis and can be immediately used as part of a quorum.

Challenges. The per-key RSM approach, however, comes with certain performance and consistency implications that we outline below.

- *Overhead of per-key consensus messages:* A coarse-grained RSM can elect a leader for a given shard and avoid the use of prepare messages for mutating operations performed on any key in the shard. In contrast, with per-key RSMs, a node would have to transmit a per-key prepare message if it had not performed the previous mutating operation on that key. Fortunately, node downtimes are low in managed environments such as ours, and a designated home node coordinates most operations on a key. We quantify the overhead associated with this using failure data collected from real deployments.
- *Reasoning about the completion status of old operations:* As discussed earlier, a coarse-grained consensus protocol such as Viewstamped Replication can discard operations initiated but not completed within a view. With fRSMs, one could perform such a view change on a per-key basis, but this would imply additional overheads even for non-mutating operations. We limit these overheads to only when a key might have outstanding incomplete operations initiated by a previous leader. (See Section 3.3.3.)

3.2 Operation API and Consistency Semantics

Operations supported: Our key-value store provides the following operations: *Create* key-value pair, *Read* value associated with a key, *Compare-and-Swap (CAS)* the value associated with a key, and *Delete* key. The *CAS* primitive is atomic: provided a key k, current value v, and a new value v', the key-value storage system would atomically overwrite the current value v with new value v'. If the current value of key k is not v, then the atomic *CAS* operation fails. Note that *Create* and *Delete* can also be expressed as *CAS* operations with a special value to indicate null objects.

We note that the *CAS* operation has a consensus number of infinity according to Herlihy's *impossibility and universality hierarchy* [12]; it means that objects supporting *CAS* can be used to solve the traditional consensus problem for an unbounded number of threads and that realizing *CAS* is as hard as solving consensus. Further, Herlihy's work shows that objects supporting *CAS* are more powerful than objects that support just reads and writes (e.g., shared registers [1]) or certain read-modify-write operations like fetch-and-increment.

We do not support blind writes, i.e., operations that merely update a key's value without providing the current value. Since all of our operations are *CAS*-like, we can provide at-most-once execution semantics without requiring any explicit per-client state as in RIFL [19]. Further, most of our updates are read-modify-write updates, so it is straightforward to express them as *CAS* operations.

Consistency model: Apart from linearizability, we aim to provide two consistency properties to simplify reasoning about operation timeouts and failures.

[1]It is worth noting that the optimization of piggybacking learn messages with subsequent commands is difficult to realize in fine-grained RSMs as a subsequent operation on the same key might not be immediate.

[2]Since we integrate the RSM consensus state into each key-value pair, we can reuse LSM APIs as well as its minor/major compaction mechanisms.

- *Session ordering:* Client operations on a given key are performed in the order in which the client issues them. This property lets a client reason about the execution status of its outstanding operations.
- *Bounded delays:* Client operations are delivered to the metadata service within a bounded delay. This property lets other clients reason about the execution status of operations issued by a failed client.

Sections 3.3.2 and 3.3.3 describe how we implement linearizable *CAS* and *read* operations using a leader-based protocol. We provide session ordering using two mechanisms: (a) leaders process operations on a given key in the order in which they were received from a client, and (b) the *read* processing logic either commits or explicitly fails outstanding operations initiated by previous leaders (see Section 3.3.3). Section 3.4 describes how coarse-grained delay guarantees from the transport layer can help clients reason about the storage state of failed clients.

Our metadata service exposes single-key operation ordering semantics as opposed to supporting transactional semantics involving multiple keys. To support multi-key operations, one can implement a client-side transaction layer that includes a two-phase commit protocol and opportunistic locking [14, 32]. This is similar to what is required of a coarse-grained RSM system to support cross-shard multi-key transactions.

3.3 Operation Processing Logic

3.3.1 Consensus State

Associated with each key is a *clock attribute* that stores information regarding logical timestamps and per-key state that is used for providing consistent updates. The clock attribute is stored along with a key-value pair in the various data structures (e.g., commit log, Memtable, and SSTables), and it comprises of the following fields.

- *epoch number* represents the generation for the key and is updated every time the key is deleted and re-created.
- *timestamp* within an epoch is initialized when the key is created and is advanced whenever the key's value is updated. The epoch number and the timestamp together represent a Paxos instance number (i.e., the sequence number of a command performed on a key-value object).
- *promised proposal number* and *accepted proposal number* associated with the key's value maintained by a given node; these represent consensus protocol state.
- *chosen bit* indicates whether the value stored along with the key represents the consensus value for the given epoch number and timestamp.

The clock attribute is a concise representation of the value associated with the key, and it is used instead of the value in quorum operations (e.g., quorum reads discussed in Section 3.3.3). Since they are frequently accessed, the clock attributes alone are maintained in an in-memory *clock cache* to minimize SSTable lookups and optimize reads/updates.

3.3.2 CAS Processing

For implementing *CAS* operations, we use a variant of the traditional Multi-Paxos algorithm, wherein we co-design different parts of the system and customize the consensus protocol for our key-value store. First, we integrate the processing associated with the consensus algorithm and the key-value store. As an example of a co-designed approach, *accept* messages will be rejected both when the promise is insufficient and when there is a *CAS error*. Second, the nodes do not maintain per-key or per-shard operation logs, but instead, skip over missed operations and directly determine and apply the accepted value with the highest associated proposal number (with a possibly much higher timestamp). Third, the processing logic speculatively updates the LSM tree and relies on subsequent operations to fix speculation errors.

Client CAS updates are built using the clock obtained via the key read previously. With each read, a client also receives the current epoch (e) and timestamp (t) for the value. The client CAS update for the key would then contain the new value along with epoch e and timestamp $t+1$. This is a logical CAS where the client specifies the new value for timestamp $t+1$ after having read the value previously at timestamp t. The request is routed to the leader of the replica group responsible for the key. It then performs the following steps.

1. **Retrieve key's consensus state:** The leader reads its local state for key k and retrieves the key's local clock. The clock provides the following values: the proposal number for a promise (p_p) and the proposal number for the currently accepted value (p_a).
2. **Prepare request:** If p_p is for a prepare issued by a different node, then the leader generates a higher proposal number, sends prepare messages to other nodes, and repeats this process until it obtains promises from a quorum of nodes. The leader skips this step if p_p and p_a are the same and refer to proposals made by the leader.

   Prepare handler: Each of the replicas, including the leader, acknowledges a *prepare* message with a promise to not accept lower numbered proposals if it is the highest prepare proposal number received thus far for the key. The replicas durably store the prepare proposal number as part of the key's clock attribute (i.e., in the commit log as well as the Memtable).
3. **Accept request:** The leader sends an *accept* message with the client-specified timestamp, i.e., $t+1$, the current epoch, and the proposal number associated with a successful prepare.

   Accept handler: At each of the replicas, including the leader, the *accept* message is processed if the current timestamp associated with the key is still t and the proposal number is greater than or equal to the local promised proposal number. If so, the key's value and the corresponding clock are recorded in the commit log and Memtable at each node. An *accept* request is rejected at one of the nodes if it has issued a promise to a higher proposal number or if the timestamp associated with the object is greater than t. In both cases, the replica returns its current value and the proposal number attached to it.

4. **Accept response processing:** The leader processes the accept responses in one of the following ways.
   - If a quorum of successful accept responses is received at the leader, the leader considers the operation to be completed and records a chosen bit on its Memtable entry for the key-value pair. It then reports success back to the client.
   - If the accept requests are rejected because the promise is not valid, then the leader performs an additional round of prepare and accept messages.
   - If the request is rejected because the (epoch, timestamp) tuple at a replica is greater than or equal to the client-supplied epoch and timestamp, then a *CAS error* is sent to the client. Further, *accept* messages are initiated to commit the newly learned value and timestamps at a quorum of nodes.

The protocol described above is faithful to the traditional consensus protocols, but it is customized for our key-value application and the use of fine-grained RSMs. In our system, a client needs to wait for a previous write to complete before issuing a subsequent write. We discuss the equivalence with coarse-grained RSM in Appendix A.4.

3.3.3 Read Processing

A read operation has to ensure the following properties upon completion: (a) the value returned should be the most recent chosen value for a given key, and (b) other previously accepted values with higher <epoch, timestamp> than the returned value are not chosen. The former requires the completion of in-progress *CAS* operations that are currently visible to the leader; this property is required for linearizability. The latter ensures that any other *CAS* operations that are in-progress but aren't visible will not be committed in the future; this is akin to a view change in the Viewstamped Replication protocol where operations that are not deemed complete at the end of a view are prevented from committing in a subsequent view.

To meet these requirements, read operations are processed in one of three different modes: *leader-only reads*, *quorum reads*, and *mutating quorum reads*. When the operation is routed to the leader, the leader checks whether it is operating in the *leader-only mode*, where all of its key-value pairs are up-to-date as a consequence of obtaining the chosen values for every key in the shard through a shard-level scan (described in Section 5.1). If the check is successful, then the leader will serve the request from its Memtable or one of the SSTables. If the leader is not operating in the *leader-only mode*, then it has to poll the replica set for a quorum and identify the most recent accepted value for a key (i.e., perform a *quorum read*). If this value is not available on a quorum of nodes, the leader has to propagate the value to a quorum of nodes (i.e., perform a *mutating quorum read*). Further, if there is an unreachable replica that might have a more recent accepted value, then the *mutating quorum read* performs an additional quorum-wide update to just the timestamp to prevent such a value from being chosen. Note that the consensus state can help determine the possibility of an update languishing in a failed/partitioned node; at least one node in a quorum set of nodes should have an outstanding promise to the failed/partitioned node, and the *read* protocol can detect this condition using a quorum operation.

We now provide additional details regarding *quorum reads* and *mutating quorum reads*. A leader not operating in *leader-only mode* satisfies a *read* request using the following steps.

1. **Quorum read request:** The leader sends the *read* request to other nodes in the replica set. Each node responds with the clock attribute associated with its local version of the key-value pair.
2. **Quorum read response:** The leader then examines the received clock attributes and checks whether any of them have a <higher epoch, timestamp> compared to the leader's clock and whether a quorum of nodes is reporting the most recent value. If the leader does not have the value associated with the highest epoch and timestamp, it obtains the value from one of the nodes reporting the most recent value. If a quorum of nodes reports not having this value, the leader propagates this value to other nodes in the quorum.
3. **Check for outstanding accepted values:** The leader then examines the received clock attributes and checks whether any of them contain a promise that satisfies the following two conditions: (1) the promise is greater than or equal to the highest proposal number associated with an accepted value, and (2) the promise is made to a node that did not respond with a clock attribute.
4. **Update timestamp to quench outstanding accepts:** If such a promise exists, then the read will perform an additional round of updates to a quorum. Let p_p be the promise associated with an unreachable node, and let v, e, and t be the value, epoch, and timestamp associated with the highest accepted proposal. The leader issues prepare commands to the replica nodes to obtain a promise greater than p_p, and then sends accept commands to the replica nodes to update their value, epoch, and timestamp fields to v, e, and $t+1$, respectively. The higher timestamp value prevents older *CAS* operations from succeeding.

The different modes for satisfying a read operation have progressively higher execution costs. In the common case, the *leader-only reads* can satisfy a read operation using local information and without communicating with the other replicas. The *quorum reads* are performed when the leader is not operating in *leader-only mode* immediately after a failover. In this case, the leader has to communicate with the other replica nodes in order to process the read request. If the most recent accepted value is not available on a quorum or if there is evidence of an unreachable node with an outstanding promise, then we resort to *mutating quorum reads* that not only incurs additional communication rounds to the replicas but also pays the overhead of writes to stable storage in order to record the updated value and timestamp. Fortunately, a *mutating quorum read* is needed only after failover and when there is an unreachable node that has obtained a promise to update the given key-value pair. Further, this is invoked only for the very first operation on the key after the failover; subsequent reads can be processed locally by the leader. This escalation of operating modes means that we incur the additional overheads associated with our use of fine-grained RSMs (e.g., per-key prepare messages and per-key timestamp updates) only in a limited number of cases.

3.4 Bounded Transport Processing

The logic outlined above allows reads to either commit or explicitly fail outstanding operations that have been received and processed by any member of the replica group. We now enhance our system to provide time bounds on the delay for propagating a command from the client to a replica node. This allows clients to also reason about the execution status of commands recently initiated by some other client in the system (e.g., the previous instance of a VM that failed unexpectedly).

CAS operations are tagged with the time at which they are initiated by the Stargate code. The leader ensures that it finishes processing the *CAS* operation within a bounded time of T seconds. If the time bound expires and the leader had failed to initiate any accept messages to process and propagate the new value, then it simply drops the request and returns a timeout message. As a consequence of this time bound, a *read* operation that is issued T seconds after an update will encounter one of the following cases: the prior update has been committed; the prior update was accepted at a subset of the nodes, in which case the read will commit it; or prior update is not at any of the responsive replicas, in which case the read will prevent the prior update from committing. The *read* can thus determine the execution status of the prior update, and repeated reads will return the same consistent value in the absence of other concurrent updates.

This bounded-time guarantee assists in handling failover of application code, migration of virtual disks across Stargate instances, and other tasks. For example, the cluster management software can delay the failover of applications until the time bound has expired to ensure that they are not affected by spurious races. For the Stargate systems code, such as that of virtual disk migration logic where stalls are not appropriate, clients directly invoke *mutating quorum read* to abort any in-flight operations from the old site until the time bound has expired.

The use of time bounds is similar in spirit to that of leases in a distributed system, and the concerns associated with the use of an implicit global clock being mitigated by the following two considerations. First, the clients of the key-value store are the block storage management services that run on the same set of nodes as the distributed key-value store and thereby share the same set of local clocks on each node. Second, in a local area enterprise cluster, time synchronization protocols such as NTP/PTP can achieve sub-millisecond time synchronization accuracy, whereas the time bounds that we provide are in the order of seconds (which is consistent with the disk timeout values in operating systems/file systems).

4 Evaluation and Deployment Measurements

Our evaluations comprise of four parts. First, we characterize the metadata service using representative traces from customer clusters. Second, we show the performance benefits of using fine-grained RSMs by comparing it with an implementation of a coarse-grained RSM (i.e., **cRSM**) approach described in Section 2. We perform these evaluations in a controlled testbed setting that runs just the metadata service and not the rest of the cluster block storage system. Note that the controlled environment has the same failure rate, request read/write ratio, and key popularity distribution that we observed in practice. Third, we present the performance of our metadata service as part of complete system evaluations. We configure a cluster with client VMs and workload generators, measure the performance of our metadata service, and characterize the performance benefits of optimizations. Finally, we report performance numbers from real-world deployments.

4.1 Experiment Setup

Our evaluations are performed on typical enterprise on-premises clusters. Specifically, our controlled testbed is a 4-node cluster, where each node is a Supermicro 1U/2U server, enclosing E5-2680 v3/E5-2620 v4 processors, 64GB/128GB DDR4 memory, two Intel DC P3600 NVMe SSDs, and a dual-port 10Gbps Intel X710 NIC. We perform the remaining evaluations on similar hardware, but at a larger scale across a large number of customer clusters. Appendix B.1 presents details of the LSM configurations that we use in practice. The replication factor for a key is three in all experiments.

4.2 Metadata Workload Characterization

We present metadata measurements from 980 customer clusters (Figure 20 in Appendix B.2). Generally, each cluster contains 3 to 30 nodes and uses 24.7TB block storage on average. The three metadata components (vDisk block, ExtentGroupID, and ExtentId) have sizes that are 0.04%, 0.12%, and 0.02% of the logical storage, respectively. Note that the size of metadata will increase when deduplication and compression are enabled due to more consolidated block storage.

Next, we characterize the metadata workload in terms of read/write ratio, value size distribution, and key access popularity by taking continuous snapshots from three custom clusters, where each cluster has at most 16 nodes. We make the following observations. First, unlike other previous key-value

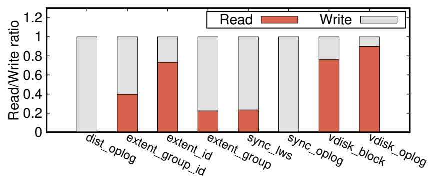


Figure 6: Read/write ratio for 8 frequently accessed metadata tables.

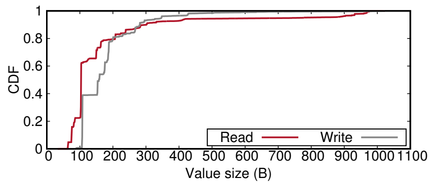


Figure 7: Value size distribution of read/write requests. Key size is less than 100 bytes.

	Cluster1	Cluster2	Cluster3
vdisk block	0.99	0.99	0.99
extent id	0.80	0.80	0.85
extent group id	0.60	0.55	0.50
extent phy_state	Uniform	Uniform	Uniform

Table 1: Key access popularity of four different metadata tables for three customer clusters. The first three types of metadata are Zipf; we show their skewness factors.

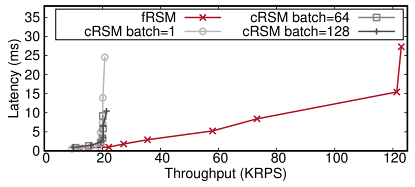


Figure 8: Latency v.s. throughput under the skewed workload for multiple shards.

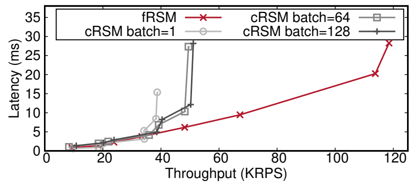


Figure 9: Latency v.s. throughput under the uniform workload for multiple shards.

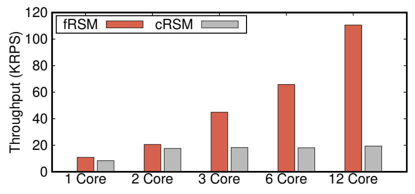


Figure 10: Maximum throu. for the skewed workload as we increase the number of cores.

store workload profiles studied under social networks/web settings [21,24], our metadata service presents various read/write ratios ranging from write-only loads for various system logs to read/write intensive ones for various filesystem metadata items (see Figure 6). Second, the read/write requests are dominated by small values, say less than 512B (see Figure 7). In fact, about 80% of reads and writes involve values that are less than 200 bytes in size. Further, requests that involve more than 1KB value sizes are about 1.0% of the reads/writes. Finally, there exist various access patterns in our metadata service. As shown in Table 1, some metadata shows highly skewed key/value accesses, while others have low skewness or even present uniform access patterns.

4.3 Benefits of Fine-grained RSMs

We now evaluate the performance benefits of fRSMs using streamlined deployments that run just the metadata service on physical nodes (as opposed to client VMs). No client workloads are executing on service nodes. We use a workload generator and configure it to issue a similar request pattern as our most frequently accessed metadata that has 43% reads and 57% writes, value size of 512B, and a Zipf distribution with skewness factor of 0.99. We also consider a uniform access case (i.e., random access pattern) as an additional workload. We inject faults into the leader using failure rate observed in the wild (Section 4.5). We evaluate fRSM and cRSM in terms of both latency and throughput.

Higher throughput. We set up a three-node replica group with twelve data shards, running across two SSDs and twelve CPU cores. In the case of cRSM, each node is a leader for four shards, each shard allocated a separate core, and six shards share each SSD. In the case of fRSM, there is a consolidated commit log striped across the two SSDs, and each operation is dynamically scheduled to a CPU core. We consider cRSM configured to perform batched commit using different batch sizes. fRSM achieves 5.6× and 2.3× higher throughputs over cRSM (with batch size of 128) for skewed and random cases, respectively (see Figures 8 and 9). This is because fRSM (1) allows requests accessing different keys to be reordered and committed as soon as they complete; (2) eliminates the computation cost associated with scanning the RSM log to identify and retry uncommitted entries; (3) avoids unnecessary head-of-line blocking caused by other requests; (4) achieves better load balance across SSDs and cores even in skewed workloads. The first three benefits can be observed even in the single shard case (Figures 11 and 12), while the next experiment further examines the load balance benefits.

Better load balancing. To examine the load-balancing benefits of fRSM , we again consider a three-node replication group with twelve data shards but vary the number of CPU cores used to process the workload. We consider the skewed workload, and we configure cRSM to use a batch size of 64. We then measured the maximum throughputs achieved and the average/p99 latency of operations when we achieve the maximum throughput (see Figures 10 and 13). fRSM provides a 1.9×, 4.1×, 6.1×, 11.0× throughput improvement and 1.9×, 2.4×, 3.3×, 5.3× (1.3×, 2.5×, 3.3×, 4.9×) avg(p99) latency reduction as we increase the number of cores from 1 core to 2, 4, 6, and 12 cores, respectively. The performance of cRSM, on the other hand, does not improve with more than two provisioned cores. Under load skews, fRSM allows balanced and timely execution of operations on different key-based RSMs, while cRSM has to commit requests in the RSM log sequentially and is subject to skews and head-of-line blocking.

4.4 Performance of Commercial Offering

We now evaluate the fRSM approach when implemented inside a commercial product providing a cluster-wide storage abstraction. This introduces many additional overheads as the metadata service is executed inside a controller virtual

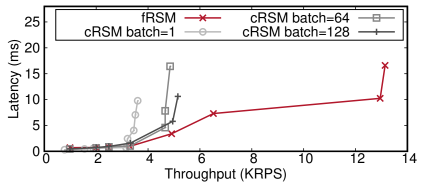


Figure 11: Latency v.s. throughput under the skewed workload for a single shard.

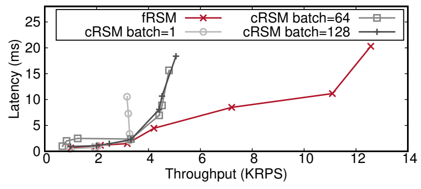


Figure 12: Latency v.s. throughput under the uniform workload for a single shard.

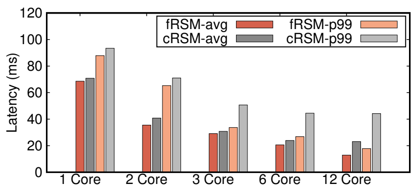


Figure 13: Average/p99 latency for the skewed workload as we increase the number of cores.

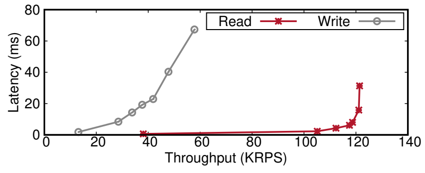


Figure 14: Latency versus throughput for reads and writes inside a Stargate cluster.

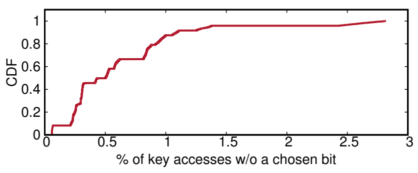


Figure 15: Operations requiring a multi-phase protocol when the leader has no chosen value.

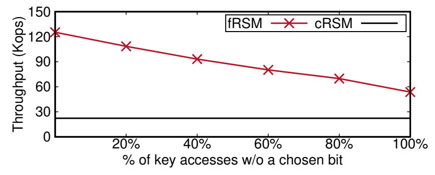


Figure 16: Throughput for the skewed workload varying the fraction of multi-phase operations.

machine, there is virtualized access to the network, and storage/CPU resources are shared with the rest of the cluster management system as well as client VMs.

We use an internal three-node cluster and an in-house workload generator that mimics various types of client VM behavior. Figure 14 reports the performance. The node is able to support peak throughputs of 121.4KRPS and 57.8KRPS for reads and writes, respectively. Under a low to medium request load, the average latency of reads and writes is 0.63ms and 1.83ms, respectively. In the appendix, we provide additional measurements of the internal cluster that quantify the benefits of using a gradation of read execution modes and utilizing the appropriate read variant for a given key. Overall, the throughput performance of fRSM inside the commercial offering is in the same ballpark as the stand-alone evaluation, but the access latency is significantly higher due to various queueing delays and interference with other storage operations that are concurrently performed by the VMs and the cluster management software.

4.5 Measurements from Real-world Deployments

High availability. We collect failure data over a two-week period (from 2018/09/12 to 2018/09/25) from about 2K customer clusters. On average, there are 70 software detached events (due to unanswered heartbeat messages) and 26 hardware failures (e.g., disk failures) per day, respectively. Crucially, our measurements show that a recovering node is able to integrate itself into the service within 30 seconds irrespective of the number of key-value operations that might have been performed when it was down. Appendix B.4 reports detailed failure handling performance. The reason for this fast recovery is that a recovering node only replays the operations in its commit log before it can participate in the consensus protocols. Each key accessed subsequently would allow the recovering node to update just that particular key-value state given the fine-grained nature of the RSMs in our system. The node can also lazily update the remaining key-value state in the background, and we observe that our system does so in about 630secs on average. In other words, the fRSM approach speeds up node integration significantly by more than 20x.

Multi-phase operations. The primary overhead associated with fRSM is the need for one or more additional rounds of protocol messages when a leader invokes an operation on a key that was previously mutated through a different leader. cRSM also incurs leadership change overheads, but they are at a shard-level, whereas fRSM incurs the overheads on a per-key basis. We quantify how often this happens in practice by measuring the fraction of instances where a leader does not have the chosen bit set and has to perform additional protocol phases. Figure 15 shows that fRSM incurs additional overheads for less than 1% of the key accesses in more than 90% of the cluster snapshots. We then performed an analysis of how the fRSM throughput degrades as we vary the number of accesses requiring multi-phase operations given the skewed workload discussed earlier. Figure 16 shows that, even though the throughput of fRSM degrades in our controlled testbed, fRSM's throughput is still higher than that of cRSM's even when 100% of the operations require multiple phases.

Cluster throughputs. We report the node/cluster throughput of the metadata layer from real deployments. Figure 17 shows the cluster throughput, where (1) every point represents a cluster data point; (2) the left y-axis represents both the throughput as well as the number of Paxos state machines that are executing per second (since every operation corresponds to a Paxos instance of a key-value pair); (3) the right y-axis is the number of nodes in the cluster. It varies from 3 nodes to a maximum of 33 nodes in the cluster; (4) the red line represents the throughput measurements per node. We can observe that our metadata layer can scale from a few thousand state machine invocations to about 393K state machine invocations per second across the cluster. The cluster with the maximum number of cluster-level operations had eight nodes,

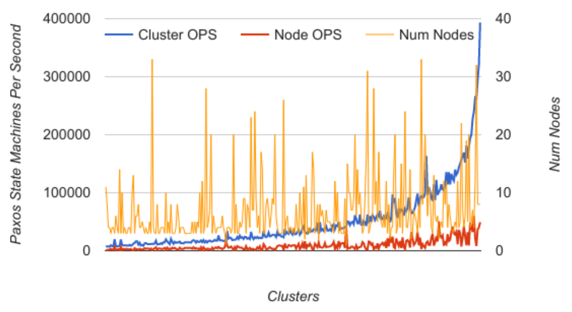


Figure 17: Cluster-level and node-level throughput for the metadata layer in the custom cluster.

and the per-node throughput is ∼59K operations per second, which is consistent with the stress tests performed on the internal cluster. Note that the peak system throughput for the other clusters could be higher, as the observed throughput is a function of the offered load.

5 Deployment-based Experience

From our experience developing the system and troubleshooting issues, we have not only improved the robustness of the system but also have learned a number of non-obvious lessons.

5.1 Fault Tolerance

Stargate provides highly-available block storage, and we describe how the metadata layer handles various cluster failures.

Transient failure. This is a typical failure scenario where the failed node recovers within a short period, e.g., a node taken offline for upgrades. When the node is the leader of a replica group, one of the other replicas will elect itself as the leader. The new leader initially processes reads and writes using quorum operations instead of transiting into *leader-only mode* (since *scan* is an expensive operation). The system keeps track of newly created SSTables on the leader and ensures these newly created SSTables are not compacted with older ones. This guarantees that new updates are segregated from older ones. When the failed node recovers, it elects itself as the leader of the replica group, provided it is the natural leader of the shard. We then transfer the newly created SSTables to the recovered node to enable it to catch up on lost updates and enter *leader-only mode* after it does so. If a significant period of time has elapsed without the failed node recovering (e.g., 30 minutes in our current system), the current leader attempts to transition to *leader-only* mode. For this, it has to *scan* the entire keyspace, by performing batched quorum reads or mutating quorum reads as necessary, to discover the up-to-date state for all keys in its shard.

Correlated or group failure. Generally, this is an uncommon event but will happen when (1) the rack UPS (uninterruptible power supply) or rack networking switch goes down; (2) the cluster undergoes planned maintenance. We apply a rack-aware cluster manager, where Stargate creates different location independent failure domains during the cluster creation and upgrade phases. Upon metadata replication, based on the replication factor (or fault tolerance level), we place replicas across different failure domains to minimize the probability that the entire metadata service is unavailable.

Optimization. It is worth noting that the choice of the LSM tree as a node's local data storage is beneficial in optimizing the handling of failures. With appropriate modifications to the LSM tree, we are able to keep the newly created data segregated. It also helps optimize the transfer of state to new nodes that are added to the replica set (to restore the replication factor in the case of persistent failures) by enabling the bulk transfer of SSTable state to the new nodes. Further, our system has a background process that periodically checks the integrity of stored data and re-replicates if necessary. This accelerates the recovery process. If a node goes down for a while, the system starts a dynamic healing approach that proactively copies metadata to avoid a two-node failure and unavailability.

5.2 Addition/Removal of Nodes

Recall that, in Stargate's metadata store, keys are spread across nodes using consistent hashing. Since we apply every update for a key to only a quorum of the key's replicas to maximize system throughput, the addition of nodes to the cluster must be handled carefully. For example, consider the addition of node A (in between Z and B) to a four-node cluster with nodes Z, B, C, and D. Say a key in the range (Z, A] has previously been written to only B and D, i.e., two out of the key's three replicas B, C, and D. Now, a read for that key could potentially return no value since two of the key's three new replicas (A, B, and C) have no record of it.

To prevent such issues, we introduce a new node by temporarily increasing the replication factor for the keys assigned to it, until the node is caught up. Having a new node catch up by issuing Paxos reads for all of its keys is, however, terribly slow; this process has taken as long as 18+ hours at one of our customers! So, we also had to develop a protocol that enables a new node to directly receive a copy of relevant portions of other nodes' SSTables. Since a new node starts serving new operations while receiving LSM state in the background, we disable caching until the new node is caught up, so as to prevent inconsistency between in-memory and on-disk state. This bulk copy method is also used during the node removal process. Besides that, we place the removed node into a *forwarding state* such that replication requests won't be accepted, but local requests will be forwarded to another node. After affected token ranges are scanned, and a quorum of the remaining nodes can respond to the request, the removed node is excised from the DHT ring.

5.3 Deletion of Keys

Consensus protocols such as Paxos are silent on the issue of deletion; it is assumed that Paxos state must be kept around forever. Therefore, when a key is deleted, correctly removing that key's Paxos state from all replicas proved to be tricky to get right for several reasons. (We describe our delete protocol in Appendix A.2.) Even after all replicas commit to their LSMs, a tombstone record indicating a key's deletion, we found that the key's old value could resurface for multiple reasons. Example causes include faulty SSDs failing to write an update to stable storage despite acknowledging having done so, or misbehaving clients issuing mutating reads with an epoch number lower than the key's epoch value when it was deleted, causing the old value to be re-propagated to all replicas. To avoid such scenarios, apart from using high-quality SSDs, we set a key's tombstone record in the LSM to be deleted only 24 hours after the third record was created. Since we use the current time to pick epoch numbers, 24 hours is sufficiently large that clock skew cannot prevent epoch numbers from monotonically increasing.

6 Related Work

Our work is related to recent research efforts in consensus protocols, consistent storage systems, metadata management, relaxed consistency, and cluster storage.

Consensus protocols: To provide consistency in the presence of node faults, we use a consensus protocol that is an extension of protocols such as Multi-Paxos [16], Viewstamped Replication [25], and Raft [26]. The crucial difference is that we integrate request processing (which in our case is read/CAS/delete operations of a key-value store) with the consensus protocol logic. This approach allows us to realize fine-grained replicated state machines that enable effective use of storage and compute units in a setting where they are scarce (since client VMs are co-located with the storage service).

Many replication protocols reduce coordination by identifying operations that can be performed independently (e.g., Generalized Paxos [17], EPaxos [23]). We employ a similar technique but use it to optimize the use of storage and computing on a server node. Our work is related to foundational algorithmic work on atomic distributed registers [1,9], but we support synchronization operations that have an unbounded consensus number (such as *CAS*).

Consistent storage systems: Our work is also related to recent work on various types of consistent key-value storage systems. Unlike Spanner [5], RIFL [19], FaRM [7], and TAPIR [35], our key-value store does not directly support transactions but rather limits itself to single key operations. Instead, it provides the atomic *CAS* primitive, which is used by the block storage management layer to make mutating updates and limited types of transactional operations. Our key-value store, however, provides bounded time operations and stronger ordering constraints that are required by legacy applications in virtualized settings. Its node-local data structures are based on those of BigTable [4] and HBase [11], and we make some modifications to aid in fast failure recovery. Our consistent storage system is also related to MegaStore [2], which provides per-row transactional updates using Paxos. Our approach integrates the Paxos algorithm with the key-value store logic in order to both enhance performance as well as provide stronger operation ordering guarantees.

Metadata management in P2P systems: Traditional DHT-based P2P storage systems (like DHash [6], Pastry [28], OceanStore [15], Antiquity [33], Ceph [34]) provide a management layer that maps physical blocks to node locations. Such metadata is a read-only caching layer that only changes when nodes join/leave. However, our metadata service maintains mappings between physical and virtual blocks, which could frequently change under VM migration. Hence, our system has a stronger consistency requirement.

Relaxed consistency: Researchers have proposed a couple of relaxed consistency models to reduce request execution latency, especially for geo-replicated key-value storage. For example, Walter [30] supports parallel snapshot isolation and conducts asynchronous replication. Within each site, it uses multi-version concurrency control and can quickly commit transactions that write objects at their preferred sites. COPS [22] is a geo-replicated key-value store that applies causal consistency across the wide area. RedBlue [20] defines two types of requests: blue operations execute locally and lazily replicate in eventually consistency manner; red operations serialize with respect to each other and require cross-site coordination. The metadata layer of our enterprise cloud storage has a linearizable requirement.

7 Conclusion

Enterprise clusters today rely on virtualized storage to support their applications. In this paper, we presented the design and implementation of a consistent metadata index that is required to provide a virtual disk abstraction. Our approach is based on using a distributed key-value store that is spread across the cluster nodes and is kept consistent using consensus algorithms. However, unlike other systems, our design uses fine-grained RSMs with every key-value pair represented by a separate RSM. Our design is motivated by the effective use of storage and computing on clusters that is achieved by flexible scheduling of unrelated operations. Our work tackles a range of challenges in realizing fine-grained RSMs and provides useful ordering guarantees for clients to reason about failures. We build and evaluate our system, compare it with coarse-grained RSMs in controlled testbed settings, and provide measurements from live customer clusters.

Acknowledgments

This work is supported in part by NSF grants CNS-1714508 and CNS-1563849. We would like to thank the anonymous reviewers and our shepherd, Rebecca Isaacs, for their comments and feedback. In addition, we thank the following people for their contributions to the initial design and implementation of the system: Radhanikanth Guturi, Mohit Aron, and Dheeraj Pandey.

References

[1] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM*, 42(1):124–142, 1995.

[2] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data System Research*, 2011.

[3] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The Primary-Backup Approach. *Distributed systems*, 2:199–216, 1993.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[5] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[6] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[7] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[8] EMC. EMC Isilon OneFS: A Technical Overview, 2016.

[9] Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. On the Efficiency of Atomic Multi-reader, Multi-writer Distributed Memory. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, 2009.

[10] Gluster. Cloud Storage for the Modern Data Center: An Introduction to Gluster Architecture, 2011.

[11] HBase Reference Guide. https://hbase.apache.org/book.html.

[12] Maurice Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[13] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, 2010.

[14] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[15] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[16] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[17] Leslie Lamport. Generalized Consensus and Paxos. Technical Report 2005-33, Microsoft Research, 2005.

[18] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and Primary-Backup Replication. In *The ACM Symposium on Principles of Distributed Computing*, 2009.

[19] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[20] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[21] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[22] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[23] Iulian Moraru, David G Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[25] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, 1988.

[26] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference*, 2014.

[27] Ohad Rodeh and Avi Teperman. zFS - A Scalable Distributed File System Using Object Disks. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2003.

[28] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2001.

[29] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[30] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[31] Sun. Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System, 2007.

[32] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[33] Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, and John Kubiatowicz. Antiquity: Exploiting a Secure Log for Wide-area Distributed Storage. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[34] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[35] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

A More Details of fRSM

In this appendix, we present additional details regarding the design of our system.

A.1 Read and CAS Algorithmic Description

Algorithm 1 READ procedure

```
procedure READ_LEADER($key$)
    if $leader\_only = 1$ then                    ▷ In leader-only read mode
        $< val_{local}, CLOCK_{local} > \leftarrow lsm\_read(key)$
        if $CLOCK_{local}.chosen$ then
            return $send\_client\_reply(key, val_{local})$
        end if
    end if
    ▷ Do a quorum read.
    for $N$ in $Replica_{group}$ do
        $send\_read(key, N)$
    end for
end procedure

procedure PROCESS_READ_REPLY_AT_LEADER($req$, $msg$)
    if $msg_{type} = SUCCESS$ then
        $++num\_responses$
    else
        $++num\_errors$
    end if
    if $CLOCK_{msg} > CLOCK_{local}$ then
        ▷ Replica has a newer value. Perform mutating quorum read.
        $write\_result \leftarrow$
            $CAS\_LEADER(key_{req}, val_{local}, CLOCK_{local}.epoch, CLOCK_{local}.ts)$
        if $write\_result = CAS_{success}$ then
            return $send\_client\_reply(key_{req}, val_{local})$
        else
            return $send\_client\_reply(key_{req}, READ_{error})$
        end if
    else if $CLOCK_{msg} = CLOCK_{local}$ then
        $++num\_responses\_with\_same\_clock$
    end if
    if $CLOCK_{msg}.P_p = CLOCK_{local}.P_p$ then
        ▷ Local leader holds promise on the replica.
        $++num\_promise\_to\_local\_leader$
    end if
    if $num\_responses\_with\_same\_clock \geq QUORUM$ then
        if $num\_promise\_to\_local\_leader \geq QUORUM$ then
            return $send\_client\_reply(key_{req}, val_{local})$
        else if $num\_responses = Replica_{group}.count$ then
            return $send\_client\_reply(key_{req}, val_{local})$
        end if
    end if
    if $num\_responses + num\_errors = Replica_{group}.count$ then
        ▷ Local leader doesn't hold promise on quorum nodes and some replicas
          didn't reply. Some of these replica nodes may have a value with higher
          timestamp, do a mutating quorum read.
        $write\_result = CAS\_LEADER(key_{req}, val_{local}, CLOCK_{local}.epoch, CLOCK_{local}.ts)$
        if $write\_result = CAS_{success}$ and
         $write\_result_{num\_responses} < Replica_{group}.count$ then
            $ts_{new} \leftarrow CLOCK_{local}.ts + 1$
            $CLOCK_{new} \leftarrow get\_clock(CLOCK_{local}.p_p, CLOCK_{local}.epoch, ts_{new})$
            $write\_result \leftarrow$
                $CAS\_LEADER(key_{req}, val_{local}, CLOCK_{new}.epoch, CLOCK_{new}.ts)$
        end if
        if $write\_result = CAS_{success}$ then
            return $send\_client\_reply(key_{req}, val_{local})$
        else
            return $send\_client\_reply(key_{req}, READ_{error})$
        end if
    end if
end procedure

procedure PROCESS_READ_AT_REPLICA($req$)
    $< CLOCK_{local} > \leftarrow lsm\_read(key_{req})$
    $send\_read\_reply(key_{req}, CLOCK_{local})$
end procedure
```

Algorithm 2 CAS procedure

```
procedure CAS_LEADER($key, val_{new}, epoch_{new}, ts_{new}$)          ▷ Client sent write.
    $< val_{local}, CLOCK_{local} > \leftarrow lsm\_read(key)$
    if $(CLOCK_{local}.epoch > epoch_{new}$ or
       $(CLOCK_{local}.epoch = epoch_{new}$ and $CLOCK_{local}.ts > ts_{new}))$ then
        return $send\_client\_reply(key, CAS_{error})$
    end if
    if $CLOCK_{local}.p_p$ is not valid then                 ▷ Issued by another node.
        $CLOCK_{new} \leftarrow get\_clock\_with\_higher\_proposal(CLOCK_{local}.p_p)$
        for $N$ in $Replica_{group}$ do        ▷ Replica group includes the leader itself.
            $send\_prepare(key, CLOCK_{new}, N)$
        end for
    else                  ▷ This leader holds the Multi-Paxos promise for this key.
        $CLOCK_{new} \leftarrow get\_clock(CLOCK_{local}.p_p, epoch_{new}, ts_{new})$
        for $N$ in $Replica_{group}$ do        ▷ Replica group includes the leader itself.
            $send\_accept(key, val_{new}, CLOCK_{new}, N)$
        end for
    end if
end procedure

procedure PROCESS_MSG_LEADER($req$, $msg$)
    if $msg_{type}$ is prepare_reply or accept_reply then
        $++num\_responses$
        if $CLOCK_{msg} > CLOCK_{req}$ or $val_{msg}$ is present then
            ▷ Replica has a higher clock or an accepted value for this clock.
            $update\_highest\_clock\_seen(key_{req}, val_{msg}, CLOCK_{msg})$
            $err \leftarrow 1$
        end if
        if $num\_responses \geq QUORM$ then
            if $err = 1$ then
                ▷ We have to run Paxos for a replica replied value
                  or with a higher clock.
                $< val_{resp}, CLOCK_{resp} > \leftarrow get\_highest\_clock\_seen(key_{req})$
                $CAS\_LEADER(key_{req}, val_{resp}, CLOCK_{resp}.epoch, CLOCK_{resp}.ts)$
                return $send\_client\_reply(key_{req}, CLOCK_{req}, CAS_{error})$
            else
                if $msg_{type}$ is prepare_reply then
                    for $N$ in $Replica_{group}$ do
                        $send\_accept(key_{req}, val_{req}, CLOCK_{req}, N)$
                    end for
                else if $msg_{type}$ is accept_reply then
                    ▷ Chosen bit is added to the local leader's memtable.
                    $send\_client\_reply(key_{req}, CLOCK_{msg}, CAS_{success})$
                end if
            end if
        end if
    else if $msg_{type}$ is prepare or accept then
        ▷ Same as the way follower works.
        return $PROCESS\_MSG\_FOLLOWER(req, msg)$
    end if
end procedure

procedure PROCESS_MSG_FOLLOWER($req$, $msg$)
    ▷ Regular Paxos protocol at the replica.
    $< CLOCK_{local}, val_{local} > \leftarrow lsm\_read(key_{req})$
    if $msg_{type}$ is prepare then
        if $CLOCK_{local} >= CLOCK_{req}$ then
            ▷ Epoch and timestamp match, value for this clock exists.
            $send\_prepare\_reply(key_{req}, CLOCK_{local}, val_{local})$
        else if $CLOCK_{local}.p_p > CLOCK_{req}.p_p$ then
            ▷ Previously accepted Promise greater than request Proposal.
            $send\_prepare\_reply(key_{req}, CLOCK_{local})$
        else
            ▷ This proposal can be accepted.
            $lsm\_write\_clock(key_{req}, CLOCK_{req})$
            $send\_prepare\_reply(key_{req}, CLOCK_{req})$
        end if
    else if $msg_{type}$ is accept then
        if $CLOCK_{req} \geq CLOCK_{local}$ then
            ▷ Received clock's epoch, timestamp, and promise are greater or
              equal to the local clock's corresponding values.
            $lsm\_write\_whole(key_{req}, val_{req}, CLOCK_{req})$
            $send\_accept\_reply(key_{req}, CLOCK_{req})$
        else
            $send\_accept\_reply(key_{req}, CLOCK_{local}, val_{local})$
        end if
    end if
end procedure
```

Algorithm 1 and 2 presents how read/CAS requests are handled at the leader and followers. They follow our protocol description in Sections 3.3.3 and 3.3.2.

A.2 Delete Processing

The key-value store also supports a delete operation, which is used by the block storage system to remove index map entries that are no longer necessary (e.g., when a virtual disk snapshot is deleted). A delete request from a client is similar to a regular *CAS* update where the client provides the epoch e and timestamp $t+1$. The leader processes a delete operation by first getting a quorum of nodes to update the value associated with the key to a special *DeleteForCell* value for epoch e and timestamp $t+1$. If the *DeleteForCell* value was not accepted by all replicas but only by a quorum, then a *DeletedCellTombstoned* message is sent to ensure replicas keep the key-value pair until the next deletion attempt. As far as the client is concerned, quorum nodes accepting a *DeleteForCell* is considered as a successful *CAS* update.

Periodically, the leader attempts to complete a two-phase deletion process to delete the value completely. When it has gotten all replicas to accept the delete request, the first phase is considered complete. It then sends a second message to instruct replica nodes to schedule the key for deletion and to remove all state associated with it. This request is recorded in *Memtable/SSTable* individually on every replica. The next major compaction on a replica will remove the state. Until then the deletion record persists at each replica with its associated clock containing epoch e and timestamp $t+1$.

Once the key deletion is successful (quorum nodes have accepted the deletion request), any new CAS updates with epoch $\leq e$ are rejected as *CAS errors*. New client updates for the key (i.e., key creation) must use a new (higher) epoch with timestamp 0.

A.3 fRSM Operation Summary

Table 2 summarizes read/write operations under various cases in terms of request latency, message count, and metadata storage operation count.

A.4 The Relationship between cRSM and fRSM

Note that fRSM works exactly the same as cRSM but in a fine-grained way. In terms of the consensus state (Figure 18 in the appendix), cRSM maintains a per-shard view number, the latest commit ID, and a log of RSM instances (where each instance has an accepted proposal number and the command value). fRSM essentially maintains information only for the most recent instance and directly encodes the promised/accepted proposal number along with the key/value pair. As a result, it doesn't require the latest commit ID. In terms of the way they handle the leader change event, cRSM uses a full two-round consensus protocol to synchronize the latest commit ID, preparing for all future commands. fRSM also takes a full two-phase consensus protocol to synchronize the consensus state for each key. In the example shown

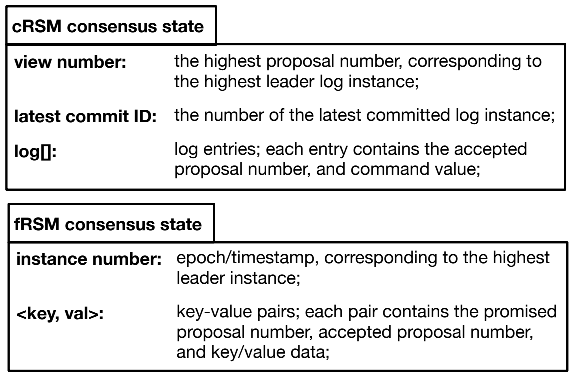

cRSM consensus state
view number:
the highest proposal number, corresponding to the highest leader log instance;
latest commit ID:
the number of the latest committed log instance;
log[]:
log entries; each entry contains the accepted proposal number, and command value;
fRSM consensus state
instance number:
epoch/timestamp, corresponding to the highest leader instance;
<key, val>:
key-value pairs; each pair contains the promised proposal number, accepted proposal number, and key/value data;

Figure 18: Consensus state comparison between cRSM and fRSM .

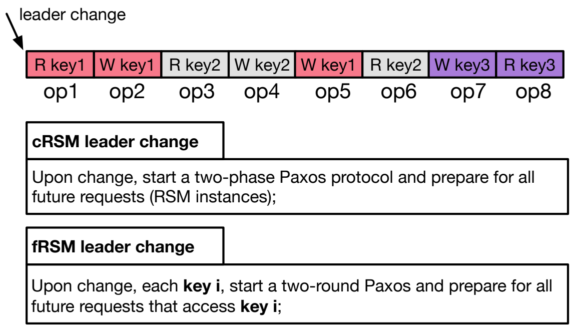

leader change
R key1
W key1
R key2
W key2
W key1
R key2
W key3
R key3
op1
op2
op3
op4
op5
op6
op7
op8
cRSM leader change
Upon change, start a two-phase Paxos protocol and prepare for all future requests (RSM instances);
fRSM leader change
Upon change, each key i, start a two-round Paxos and prepare for all future requests that access key i;

Figure 19: Leader change comparison between cRSM and fRSM .

in Figure 19 (in the appendix), where there are eight operations accessing three different keys, cRSM issues the leader prepare message at op1, while fRSM performs this prepare at op1, op3, and op7.

B More Real-world Evaluation

B.1 LSM Configuration

Table 3 shows key LSM parameters. They are configured based on the physical storage media, cluster setup, and metadata characteristics. The table presents the default values.

B.2 Deployment Scale

Figure 20 presents the deployment scale in terms of node number, storage size, and metadata size.

B.3 Internal Cluster Measurements

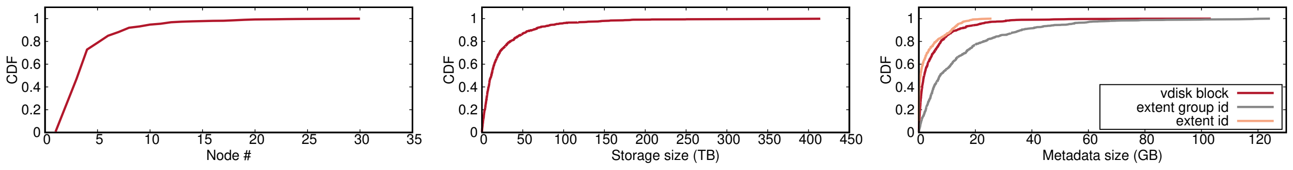
We consider again the internal cluster running the complete storage and virtualization system along with client VMs invoking stress tests on the metadata and storage layer (as discussed earlier in). We report the average/p99 latency distribution of read/write requests (Figure 21), showing comparable end-to-end performance for read and write operations. We also evaluate the performance of leader-only reads. Leader-only mode significantly reduces the number of protocol messages and storage accesses, enabling fast metadata access. Figures 22 and 23 show that leader-only mode results in benefits across different value sizes. On average, across various sizes,

Operations	Latency (RTT)	Message #	Leader LSM RD. #	Leader LSM WR. #	Follower LSM RD. #	Follower LSM WR. #
Cold CAS	2	$4\lceil n/2 \rceil$	1	2	2	2
Warm CAS	1	$2\lceil n/2 \rceil$	1	1	1	1
Leader-only Read	0	0	1	0	0	0
Quorum Read	1	$2n$	1	0	1	0
Mutating Quorum Read	3	$2n+4\lceil n/2 \rceil$	2	2	3	2

Table 2: Message RTTs and LSM read/write counts with no cache for the leader and the follower under different settings. *n* is the number of replicas. *cold CAS* refers to the case that the proposal of a key is issued by another node so that the leader has to invoke the two round Paxos. *warm CAS* means that the leader is able to skip the 1st round of prepares.

Parameter	Description	Default value
max_heap_size	Maximum heap size of the metadata store	2∼4GB
flush_largest_memtables	Heap usage threshold when flushing the largest memtable	0.9
default_memtable_lifetime	Life time in minutes for any memtable	30
min_flush_largest_memtable	Minimum memtable size forced flush when heap usage is high	20MB
max_commit_log_size_on_disk	Maximum disk usage by commit logs before triggering a cleanup task	1GB
commitlog_rotation_threshold	Maximum size of an individual commit log file	64MB
number_of_compaction_threads	Number of threads to perform minor/major compaction	2
compaction_throughput_limit	Maximum disk throughput consumed by compaction on a disk	64MB

Table 3: LSM performance-sensitive parameters.



(a) Node #. (b) Storage size. (c) Metadata size.

Figure 20: Node #, storage size, and metadata size CDF across 980 custom clusters.

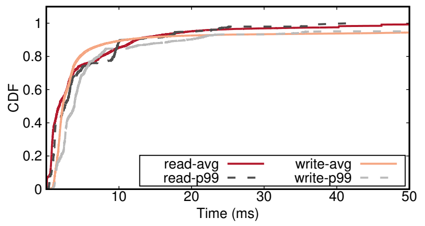


Figure 21: Average/p99 read/write latency CDF inside a Stargate cluster.

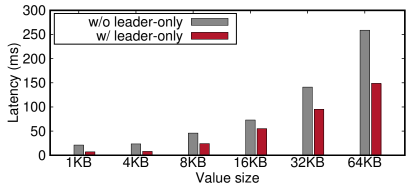


Figure 22: Latency versus value size, compared between with and without leader-only mode.

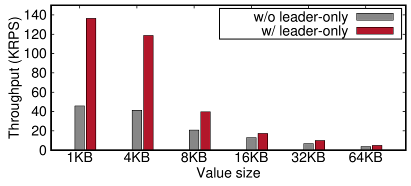


Figure 23: Throughput versus value size, compared between with and without leader-only modes.

leader-only mode halves the latency and more than doubles the throughput. This underscores the benefits of using a gradation of read execution modes and utilizing the appropriate read variant for a given key.

B.4 Failure and Recovery Measurements

We provide additional details on the failure and recovery measurements from our customer clusters. Figure 24 shows the number of software detached events and fatal hardware errors across the measurement period across all of the 2K clusters. Both of them are detected by the DHT health manager. Under software failures, our system will quickly restart the metadata service and rejoin the DHT ring, consuming 2.7s on average. Upon fatal hardware errors, we reboot the server box and then walk through some device checks (e.g., storage media and network). Figure 27 presents our observed server downtime distribution. After node failure, the system follows a 3-phase node handling failure to recover to the leader-only mode, i.e., regaining leadership (*T1*), performing local recovery (*T2*), and performing a leader scan (*T3*). Based on our collected traces, we observe that *T1* consumes 1.0ms. During *T2* phase, the node reads the commit log and executes missing requests. Figures 25 and 26 present the CDF of local node recovery (*T2*) and the number of recovered operation records (from the committed log) for 4 clusters, respective. Note that our cluster node is able to serve client requests starting from *T2* in a non-leader-only mode and enters the leader only mode after the scan finishes (*T3*).

The duration of the *T3* phase depends on scan performance. To enable leader-only reads, the new leader must scan through its owned range to learn the latest values. In some cases, Paxos writes must be done, and this imposes additional la-

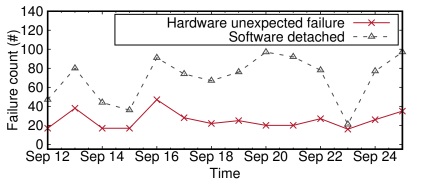


Figure 24: Failure rate among 2K custom VMs (Year 2018).

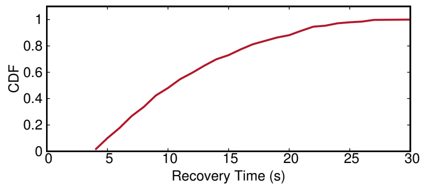


Figure 25: CDF of node recovery time inside a Stargate cluster.

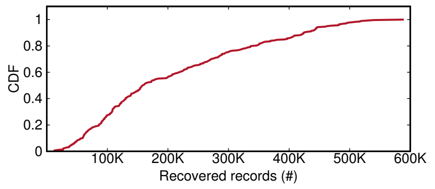


Figure 26: CDF of recovered records inside a Stargate cluster.

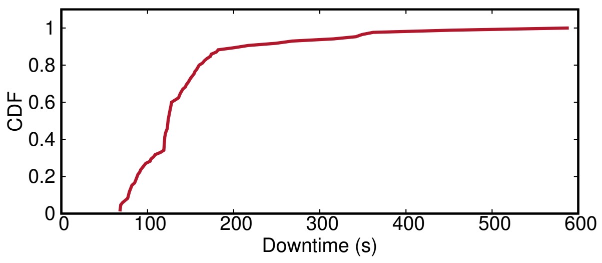


Figure 27: Node downtime CDF (after the hardware failure).

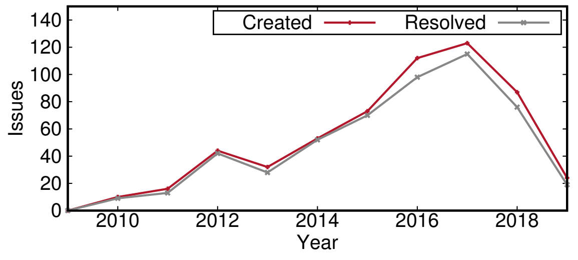


Figure 28: Metadata service corruption report over years.

tency costs for the scanning process. The worst-case repair time occurs when Paxos operations must be performed for every key. Conversely, the best-case scan time occurs when no consensus operations need to performed (i.e., the node has all of the latest data). Figure 29 provides the time associated with scans when the nodes are loaded with data comprising of 32-byte keys and 8KB values. When repairs need to run for every value, the total scan time is about 6× long. These measurements show the quick integration of recovering nodes into the metadata service.

B.5 Metadata Corruption Reports

Figure 28 shows the number of cases that have been reported by our QA based that have caused data unavailability or corrupt data being returned to the client based on the tests. We have not culled for duplicate issues, where the single cause manifested in multiple ways. The broad category of failures has changed over the years. Initially, it was the interaction with the local filesystems (fsyncs, o_directs), persistent media corruption, cluster misconfiguration. In recent years, it has been due to the addition of new features like leader-only reads, fast range migrations, balancing with no node downtimes. There have been a handful of protocol implementation issues that were weeded out fairly quickly.

C Testing Framework

Our testing strategy and framework has evolved over the years. Based on experience, we have found white box testing to be one of the key ways to identify implementation issues in new features and avoid regressions. We have instrumented code to simulate various scenarios and probabilistic error conditions like replica packet drops, timeouts, and erroneous key states. Whenever a bug is discovered in the field or in black box testing, we add a white box test to simulate the same condition

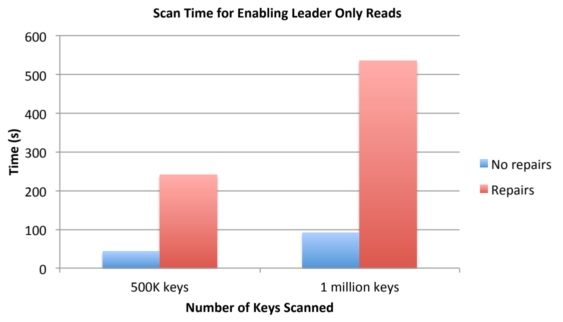


Figure 29: Time to perform a scan in order to enable leader-only reads.

along with making the fix.

We also have multiple test clusters that do end-to-end black box testing with error injections. Errors can be in the form of service restarts, service down, corruption over the network, timeouts, replays, and corruption of persistent store. As an example, we use a test devised specifically for fRSM that performs atomic increments on value(s) stored in key(s) *n* times (where each of the atomic increment is a CAS update) and, at the end, when all clients are done, check whether the final value of each key is *n* times the number of clients. While these clients are incrementing values using CAS, we randomly kill replica/leader nodes, insert failures, randomly drop messages between leader/replica nodes, add a delay in replying to messages, etc. Apart from incrementing values in the keys, we also delete keys in the middle of the test to go through the delete workflow and re-insert the key(s) with the value(s) seen just before the deletes, so the clients can continue incrementing the values. We also add/remove replicas to the metadata service while this test is underway to test add/remove node scenarios and different read variants. These type of tests can be performed within a developer environment and have aided in building a robust system.

It is non-trivial to pinpoint performance bottlenecks due to the complexity of our system. We instrument our logic across the metadata read/write execution path and report runtime statistics at multiple places, such as the number of outstanding requests at the Paxos leader, the hit rate of the key clock attribute, read/write/scan latency at leader and follower of one key-range, etc. This instrumentation has been helpful in identifying various performance issues.

Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log

Cong Ding David Chu Evan Zhao Xiang Li[†] Lorenzo Alvisi Robbert van Renesse
Cornell University [†]*Alibaba Group*

Abstract

The *shared log paradigm* is at the heart of modern distributed applications in the growing cloud computing industry. Often, application logs must be stored durably for analytics, regulations, or failure recovery, and their smooth operation depends closely on how the log is implemented. *Scalog* is a new implementation of the shared log abstraction that offers an unprecedented combination of features for continuous smooth delivery of service: Scalog allows applications to customize data placement, supports reconfiguration with no loss in availability, and recovers quickly from failures. At the same time, Scalog provides high throughput and total order.

The paper describes the design and implementation of Scalog and presents examples of applications running upon it. To evaluate Scalog at scale, we use a combination of real experiments and emulation. Using 4KB records, a 10 Gbps infrastructure, and SSDs, Scalog can totally order up to 52 million records per second.

1 Introduction

A shared log[1] offers a simple and powerful abstraction: a sequence of ordered records that can be accessed and appended to by multiple clients. This combination of power and simplicity makes them a popular building block for many modern datacenter applications. All cloud providers offer a shared log service (e.g., AlibabaMQ [2], Amazon Kinesis [3], Google Pub/Sub [8], IBM MQ [10], Microsoft Event Hubs [13], and Oracle Messaging Cloud Service [14]), which is also available through multiple open source implementations (e.g, Apache Kafka [36], Corfu [21], and FuzzyLog [41]).

Shared logs are used to (1) record and analyze web accesses for recommendations, ad placement, intrusion detection, performance debugging, etc. [11, 31, 36, 50]; (2) prepare a transport between stages in a processing pipeline that may be replayed for failure recovery [11, 50], and more broadly; (3) address the trade-off between scalability and consistency [51]. Consider, for instance, deterministic databases [22, 34, 35, 47, 48]: retrieving transactions from a single shared log allows these databases to shorten or eliminate distributed commit protocols, avoid distributed deadlocks, and achieve, in principle, superior transactional scalability [44, 47, 48].

An ideal implementation of the shared log abstraction should be capable of growing elastically in response to the needs of its client applications, without compromising availability; recover quickly from failures; adopt the data layout that best matches the performance requirement of its clients; and scale write throughput without giving up on total order. Unfortunately, no single shared log today can offer this combination of features. In particular, no shared log provides both total order and *seamless reconfiguration*, i.e., the capability to reconfigure the service without compromising its global availability.

The state-of-the-art Corfu [21] can adapt to applications' needs by adding or removing storage servers, while maintaining total order across records stored on different servers. However, any change in the set of storage servers makes Corfu unavailable until the new configuration has been committed at all storage servers and clients. The Corfu data layout is defined by an inflexible round-robin policy, with significant performance implications: for example, reads require playing back the log where relevant updates are interspersed with unrelated records (a potential performance bottleneck) and writes cannot be directed to the closest storage server to reduce latency. vCorfu [51], an object store based on Corfu, addresses the issue of slow reads by complementing the Corfu shared log with *materialized streams*, log-like data structures that store together updates that refer to the same object. For this gain in performance, vCorfu pays in robustness: whenever a log replica and a stream replica fail concurrently—a more likely event as the system scales up [25]—vCorfu is at risk of losing data. Finally, the tension between scaling across multiple storage servers and guaranteeing total order ultimately limits Corfu's write throughput. The optimized Corfu implementation used in Tango [22] achieves, to the best of our knowledge, the best throughput among today's totally ordered shared logs but, at about 570K writes/sec, its performance falls short of the needs of applications like Taobao [16], Alibaba's online market, which ran millions of database writes/sec at its 2017 peak [1].

[1]A shared log is also known as a *message bus*, but not all message buses provide a durable message store.

Scalog, the new shared log that this paper introduces, aims to address these limitations. Like Corfu, Scalog can scale horizontally by adding *shards* and guarantees a single total order for all records, across all shards. However, reconfiguring Scalog by adding or removing shards requires no global coordination and does not affect its availability. Further, Scalog's API gives applications the flexibility to select which records will be stored in which shard: this allows Scalog to replicate the functionality offered by vCorfu's materialized streams without trading off robustness. Indeed, Scalog operates under weaker failure assumptions (and hence is inherently more robust) than prior totally-ordered shared logs [21,22,51]: it assumes that faulty servers will *crash* [29], rather than *fail-stop* [45], thus sidestepping the so-called "split-brain syndrome" [26].[2] Finally, though Scalog cannot scale write throughput indefinitely, it can deliver throughput almost two orders of magnitude higher than Corfu's, with comparable latency.

Scalog's properties derive from a new way of decoupling global ordering from data dissemination. Decoupling these two steps is not a new idea. For example, in Corfu, the sequencer that globally orders records is not responsible for their replication; once the order has been decided, replication is left to the clients, thus allowing data dissemination to scale until ordering ultimately becomes the bottleneck.

The key to Scalog's singular combination of features is to turn on its head how decoupling has traditionally been achieved. In Corfu (as well as Facebook's LogDevice [12]), order comes before persistence: records are first assigned unique sequence numbers in the total order, and then replicated; in Scalog, the opposite is true: records are first replicated, and only then assigned a position in the total order.

As mentioned above, Corfu requires that all clients and storage servers hold the same function to map sequence numbers to specific shards, causing Corfu to be temporarily unavailable when shards are added or removed. By ordering only records that have already been replicated, Scalog sidesteps the need to resolve the delicate case in which a client, having reserved a slot in the total order for one of its records, fails before making that record persistent. Without this burden, Scalog can seamlessly reconfigure without any loss of availability, and give applications the flexibility to independently specify which shards should store their records, thus matching vCorfu's data locality without the need of dedicated stream replicas.

Specifically, Scalog clients write records directly to storage servers, where they are (trivially) FIFO ordered without the mediation of a global sequencer. Records received by a storage server are then immediately replicated across the other storage servers in the same shard. To produce a total order, Scalog periodically interleaves the FIFO ordered sequences of records stored at each server.

[2]The essential difference is that crash failures cannot be accurately detected, while in the fail-stop model, it is assumed that failures can be detected accurately by some oracle. Violation of this assumption can lead to inconsistencies.

We recognize that not all applications require a global total order, and many industrial applications have been built using shared logs such as Kafka [36] that only provide a total order per shard. However, our unique way of providing total order comes at practically no cost to throughput even under reconfiguration, while latency within a datacenter is no more than a few milliseconds. Programmers thus only need to consider performance when deciding how to shard their applications, an otherwise difficult balancing game between achieving the required throughput and correctness [18, 27, 28, 44]. Also, a global total order supports reproducibility, simplifying finding bugs in today's complex distributed applications.

Our evaluation of a Scalog prototype implemented on a CloudLab cluster [4] confirms that Scalog's persistence-first approach comes closer to an ideal implementation of the shared log abstraction in three main respects:

- It provides seamless reconfiguration. Our prototype sees no increase in latency or drop in throughput while Scalog is being reconfigured.
- It offers applications the flexibility to select where the records they produce should be stored. We use this capability to build vScalog, a Scalog-based object store that matches the read latency of vCorfu (and achieves twice its read throughput) while offering stronger fault-tolerance guarantees.
- It offers (almost) guilt-free total ordering of log records across multiple shards. While Scalog does not eliminate the trade-off between scalable write throughput and total order, it pushes the pain point much further: its maximum throughput is essentially limited by the maximum number of shards times the maximum throughput of each shard. With 17 shards, each with two storage servers, each processing 15K writes/sec, our prototype achieves a total throughput of 255K totally ordered writes/sec. Through emulation, we demonstrate that, at a latency of about 1.6 ms, Scalog can handle about 3,500 shards, or about 52M writes/sec—two orders of magnitude higher than the best reported value to date at comparable latency [22].

2 Motivation and Design

We motivate Scalog and its design principles with an online marketplace that enables sellers to list and advertise their merchandise, and allows buyers to browse and purchase. Page views and purchases are stored in a log that is used for multiple purposes, including extracting statistics (such as the number of unique visitors), training machine learning algorithms that can recommend merchandise and sellers to buyers, and simplifying fault tolerance by providing applications with a shared "ground truth" about the system's state.

This example highlights six key requirements for the underlying shared log. First, the log requires *auto-scaling*, the

`append`(r)	Append record r, and return the global sequence number.
`trim`(l)	Delete records before global sequence number l.
`subscribe`(l)	Subscribe to records starting from global sequence number l.
`setShardPolicy`(p)	Set the policy for which records get placed at which storage servers in which shards.
`appendToShard`(r)	Append record r, and return the global sequence number and shard identifier.
`readRecord`(l,s)	Request the record with sequence number l from shard s.

Table 1: Scalog API

ability to dynamically increase or decrease available throughput as needs change (e.g., during the holiday seasons) without causing any downtime. Second, for reproducibility during debugging and consistent failure recovery, the log should be totally ordered. Third, the log must minimize latency, for example, by allowing log clients to write to the nearest replica. Fourth, the log must provide high append throughput to support large volumes of store activity. Fifth, the log must provide high sequential read throughput to support analytics, which periodically read sub-sequences of the log. Finally, the log must be fault tolerant, as the online services that depend on it should be uninterrupted. Below, we discuss how Scalog addresses these requirements.

2.1 Scalog API

Scalog provides the abstraction of a totally ordered shared log. Table 1 presents a simplified API that omits support for authentication and authorization, as well as the ability to subscribe only to records that satisfy a specific predicate.

The first three methods are sufficient for most applications. The `append` method adds a record to the log. When it returns, the client is guaranteed that the record is *committed*, meaning that it cannot be lost (it has been replicated onto multiple disks) and that it has been assigned a *global sequence number* (its unique log position among committed records). The `trim` method allows a prefix of the log to be garbage collected. Finally, the `subscribe` method subscribes to committed log records starting from global sequence number l. Scalog guarantees that (1) if `append`(r) returns a sequence number, each subscriber will eventually deliver r; and (2) any two subscribers deliver the same records in the same order.[3] Note that these guarantees are sufficient to implement a replicated state machine [46] using Scalog.

To achieve high throughput and support flexible allocation of resources, Scalog structures a log as a collection of *shards*, each in turn containing a collection of records. Applications can exploit the existence of shards to optimize performance with the remaining three API methods. The `setShardPolicy` method lets applications specify a function used to assign records to storage servers and shards. The `appendToShard` method behaves as `append`, but in addition returns the identifier of the shard where the record is stored. The `readRecord` method allows random access to records by sequence number, assuming the shard identifier is known (e.g., for having been returned by a prior invocation of `appendToShard`). Under concurrent access, Scalog offers fully linearizable semantics [32]—the strongest possible consistency guarantee.

Besides this API, Scalog provides various management interfaces that allow reconfiguring the log seamlessly in response to failures and to the changing needs of the applications it serves. Specifically, Scalog can create new shards on-the-fly (if load increases), as well as turn shards from *live* to *finalized*. New records can only be appended to live shards; once finalized, a shard is immutable.

Finalizing shards serves three purposes. First, Scalog optimizes finalized shards for read throughput. To prevent read-heavy analytics workloads from affecting the performance of online services, an operator may create new shards, finalize the old shards (effectively, creating a checkpoint), and then run the analytics workload on the finalized shards. Second, when a storage server in a shard fails, append throughput may be affected; rather than recovering the failed server, Scalog allows finalizing the entire shard and replacing it with a new one (see §3). If one wishes to restore the level of durability, additional replicas may be created after a shard is finalized. Third, finalized shards may be garbage collected: this is how resources are reclaimed after a log is trimmed.

2.2 "Order first" Considered Harmful

Current totally ordered log implementations [6, 12, 21, 22] share a similar architecture: to append a new record, a client first obtains the record's position in the log, the *log sequence number*, via some *sequencer*, and then proceeds to make the record persistent. This design raises two challenges.

The first challenge is supporting flexible data placement and seamless reconfiguration. The difficulty comes from having to maintain the consistency of the log when failures occur—a dilemma that arises whenever a storage system makes decisions about an item's metadata before making persistent the item itself [20,30]. Under failure, a record may get lost; this "hole" in the log needs to be filled before other tasks can read beyond the missing entry. Solving this problem requires a costly system-wide agreement on a mapping from log sequence numbers to where records are stored: changes to this mapping are exceedingly expensive, since, until a new mapping is agreed upon, the system cannot operate. For applications using the log, this cost translates into two main limitations. First, they have no practical way of dynamically optimizing the placement of the records they generate, since frequent changes to the mapping would be prohibitively ex-

[3]These guarantees hold only in the absence of trimming. Trimmed records may never be delivered to some subscribers.

pensive; second, each time storage servers are added or removed for any reason, they experience a system-wide outage until the new mapping is committed and distributed [21, 22].

The second challenge is that the sequencer can quickly become a bottleneck: designing a sequencer capable of operating at high throughput requires significant engineering effort, frequently involving custom hardware, such as programmable switches (e.g., NOPaxos [40]) or write-once disks (e.g., Corfu [21]).

2.3 Scalog Design Overview

By adopting a persistence-first architecture, Scalog avoids these challenges. It achieves no-downtime reconfiguration, quick failure recovery, and high throughput, without using custom hardware, via a new, simple, protocol for totally ordering persistent records across multiple shards.

In Scalog, each shard is a group of storage servers that mutually replicate each other's records. Scalable throughput is achieved by creating many high-throughput shards, as in Kafka [50]; however, unlike Kafka, which only provides total order within individual shards, Scalog delivers a single total order across all shards.

Persistence in Scalog is straightforward. A client sends a record to a storage server of its choice, before knowing the record's global sequence number. Storage servers append incoming records, which may come from different clients, to a *log segment* which they replicate by forwarding new records through FIFO channels to all other storage servers within their shard. Thus, each storage server maintains a *primary log segment* as well as backup log segments for every other storage server in its shard. Because of FIFO channels, every backup log segment is a prefix of the primary log segment.

Scalog's second key insight is leveraging the FIFO ordering of records at each storage server to leapfrog the throughput limits of traditional sequencers.

Periodically, each storage server reports the lengths of the log segments it stores to an *ordering layer*. The ordering layer, also periodically, determines which records have been fully replicated and informs the storage servers. Using the globally ordered sequence of reports from the ordering layer, a storage server can interleave its log segments into a global order consistent with the original partial order. Afterward, the storage server can inform clients that their records are both durably replicated and totally ordered.

The ordering layer of Scalog interleaves not only records but also other reconfiguration events. As a result, all storage servers see the same update events in the same order. When a storage server in a shard fails, Scalog's ordering layer will no longer receive reports from the storage server and naturally exclude further records. Other shards are not affected. Thus, clients connected to a shard containing a faulty storage server can quickly reconnect and send requests to servers in other shards. Concurrently, the affected shard is finalized.

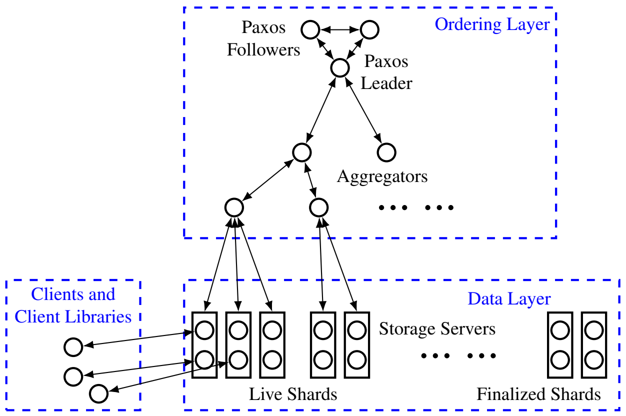


Figure 1: Scalog's architecture: arrows denote communication links; circles denote servers; each rectangle denotes one shard. Servers in the same shard communicate with each other. In this example, both shards and the Paxos instance in the ordering layer are configured to tolerate one crash.

3 Architecture

Figure 1 presents an overview of Scalog's architecture, highlighting its three components: a *client library*, used to issue `append`, `subscribe`, and `trim` operations; a *data layer*, consisting of a collection of shards, storing and replicating records received from clients; and an *ordering layer*, responsible for totally ordering records across shards.

Client Library. The library implements the Scalog API and communicates with the data layer (see Table 1).

Data Layer. Scalog's data layer distributes load along two dimensions: each log consists of multiple shards, and each shard consists of multiple storage servers. Each storage server is in charge of a *log segment*. Clients send records directly to a storage server within a shard. When a storage server receives a record from a client, it stores the record in its own log segment. For durability, each server replicates the records in its log segment onto the other storage servers in its shard. To tolerate f failures, a shard must contain at least $n = f + 1$ storage servers.

Ordering Layer. The ordering layer periodically summarizes the fully replicated prefix of the primary log segment of each storage server in a *cut*, which it then shares with all storage servers. In a Scalog deployment with m shards, each comprising n storage servers, the cut has $m \cdot n$ entries, each mapping a storage server identifier to the sequence number of the latest durable record in its log segment. The storage servers use these cuts to deterministically assign a unique global sequence number to each durable record in their log segments. Besides enabling global ordering, the ordering layer is also responsible for notifying storage servers of reconfigurations.

The ordering layer must address two concerns: fault tolerance and scalability under high ordering load. Scalog addresses the first concern by implementing the ordering layer logic using Paxos [38]. The second concern is that the over-

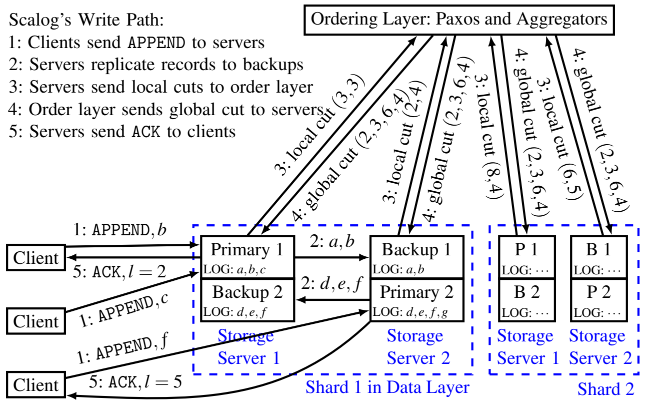


Figure 2: Scalog message flow for `append` operations

head of managing TCP connections and handling the ordering requests could overwhelm the ordering layer when there are a large number of shards. This concern is addressed with the help of the aggregators, illustrated in Figure 1. Scalog spreads the load using a tree of aggregators that relay ordering information from the storage servers at the leaves up to the replicated ordering service. Each leaf aggregator collects information from a subset of storage servers (we assume servers in the same shard report to the same leaf aggregator) and determines the most recent durable record in their log segment before passing the information up. Aggregators use soft state and do not need to be replicated—if suspected of failure, they can easily be replaced. The ordering reports passed up the tree are self-sufficient, and need not be delivered in FIFO order. The aggregator tree is maintained by the replicated service in its root—no decentralized algorithms are needed to eliminate loops and orphans.

4 Scalog's Workflow

To further elucidate how Scalog works, we present a detailed explanation of the execution paths for *append*, *read*, and *trim* (garbage collection) operations.

4.1 Append Operations

When an application process first invokes its client library to start appending data to the log, the client library chooses a shard according to the current sharding policy set by `setShardPolicy`(p). If no policy has been specified, Scalog applies its default selection policy, choosing a random storage server in a random live shard as the write target.

Having established a destination shard s and storage server d, the application process can add records to the log. When `append`(r) or `appendToShard`(r) is invoked, the client library forwards record r to storage server d in an `APPEND` message (Figure 2, Step 1) and awaits an acknowledgment.

As shown in Step 2 of Figure 2, each storage server replicates in FIFO order the records it receives onto its peer storage servers in s. In-shard replication resembles Primary-Backup (PB) [19, 24]: each storage server acts as both Primary for the records received directly from clients and Backup for the records in the log segments of its peer storage servers in the shard. Scalog differs from PB in how storage servers learn which records in their log segment have become durable. Instead of relying on direct acknowledgments from its peers, each storage server periodically reports to the ordering layer a *local cut*—an integer vector summarizing the records stored in this storage server's log segments (Step 3 in Figure 2). Because log segment replication occurs in FIFO order, each integer in the local cut is an accurate count of the number of records stored in the corresponding log segment.

The ordering layer combines these local cuts to determine the latest durable record in the log segment of each storage server. Let v_i be the local cut for server i in shard s; $v_i[i]$ represents the number of records in i's log segment (i.e., those that i, serving as Primary, received directly from its clients) while $v_i[j], j \neq i$, is the number of records that server i is backing up for its peer storage server j. The ordering layer can then compute the number of durable records in i's log segment as the element-wise minimum of all $v_j[i]$ for all storage servers j in s. For instance, assume $f = 1$ and suppose the ordering layer has received from the two storage servers r_1 and r_2 in shard s the local cuts $v_1 = \langle 3,3 \rangle$ and $v_2 = \langle 2,4 \rangle$. Then, $\langle 2,3 \rangle$ expresses all durable records in shard s (see Shard 1 in Figure 2). By repeating this process for all storage servers in every shard, the ordering layer assembles a *global cut*, a map that represents records stored in all log segment replicas and therefore durable. To prevent the number of entries in global cuts from growing indefinitely, we use a single integer to represent the total number of records in all finalized shards. The ordering layer then forwards each global cut to all storage servers (Step 4 in Figure 2).

The totally ordered sequence of cuts can be used to induce a total ordering on individual records. Summing the sequence numbers in the elements of a cut gives the total number of records that are ordered up to and including that cut. The difference between any two cuts determines which records are covered by those two cuts. We use a deterministic function that specifies how to order the records in between two consecutive cuts. In our current implementation, we use a simple lexicographic ordering: records in lower-numbered shards go before records in higher-numbered shards, and within a shard records from lower-numbered storage servers go before records from higher-numbered storage servers.

Therefore, upon receipt of a global cut, a storage server can determine which records in its primary log segment are now globally ordered, and then acknowledge the corresponding append requests by returning the record's global sequence number to the client (Step 5 in Figure 2). Should a storage server fail, a client can ask any of its backup for the current status of its records.

Note that the load on the ordering layer is independent of the write throughput—it only depends on the number of storage servers and the frequency of their reports.

4.2 Read Operations

Applications can read from the log either by subscribing or by requesting specific records. The `subscribe` operation broadcasts the request to a random storage server in each shard. Upon receiving a `subscribe`(l) request, the storage server sends the client all records it already stores whose global sequence number is at least l and then continues forwarding future committed records.

Recall that an application process, by calling `appendToShard`(r), obtains the shard identifier s that stores record r, as well as its global sequence number l. If at a later time the process needs to bring r back in memory, it can do so by invoking `readRecord`(l,s). In response, the client library contacts a random storage server in s to read the record associated with global sequence number l. The receiving storage server then computes $l_{\max}$, the largest global sequence number it has observed, by applying the deterministic scheme of §4.1 to the latest cut received from the ordering layer, and proceeds to compare l and $l_{\max}$. If $l \leq l_{\max}$, the storage server uses l to look for r in its local log and, if it finds it, returns it; otherwise, if the record has been trimmed (see §4.3), it returns an error. If $l > l_{\max}$, the storage server waits for new cuts from the ordering layer and updates $l_{\max}$ until $l \leq l_{\max}$; only then does it proceed, as in the previous case, returning to the application process either r or an error message. Allowing responses only from storage servers for which $l \leq l_{\max}$ is critical to guarantee linearizability for concurrent read and append operations, as it prevents stale storage servers from incorrectly returning error messages. The client library may, however, timeout, waiting for a storage server to respond; if so, the client library contacts another storage server in s (the storage server holding r in its log segment is guaranteed to eventually respond).

4.3 Trim Operations

Calling `trim`(l) garbage collects the log prior to the record with global sequence number l. The client library broadcasts the `trim`(l) operation to all storage servers in all shards; upon receipt, they proceed to delete the appropriate prefix of the log stored in their respective log segments.

4.4 Reconfiguration and Failure Handling

Reconfiguration can happen often in Scalog, not only to recover from failures (which are more likely as scale increases), but also to handle growing throughput or needed capacity. For example, an application that needs to run a read-intensive analytics job can finalize the shards storing the relevant data, making them read-only. For these reasons, Scalog strives to make adding and finalizing shards seamless.

4.4.1 Adding and Finalizing Shards

Adding a new shard is straightforward: as soon as the shard and its servers register with the ordering layer, the new shard can be advertised to clients. Other shards are unaffected, but for the larger-sized cut, its storage servers will henceforth receive from the ordering layer.

We distinguish two types of shard finalization: scheduled finalization and emergency finalization. Scheduled finalizations are initiated in anticipation of shard workload changes. To transition clients off of shards facing impending finalization, Scalog supports a management operation that causes the ordering layer to stop accepting ordering reports from a shard after a configurable number of committed cuts. This gives clients a "grace period" so that they can smoothly transition to another live shard. Emergency finalizations are needed when a server in a shard fails (see *Finalize & Add* in §4.4.2); these failed shards are finalized immediately.

4.4.2 Handling Storage Server Failures

Failing or straggling storage servers are detected either by Paxos servers directly connected to them or by aggregators. Problems are notified to the ordering layer, which in turn initiates reconfiguration. Applications have three options to configure how Scalog handles slow or failed storage servers.

Finalize & Add (Requires at least $f+1$ storage servers per shard): If a storage server is suspected of having failed or is intolerably slow, its entire shard s is finalized. Clients of s can redirect their writes to other shards; concurrently, a new shard is added to restore the log's overall throughput. Because the ordering layer totally orders finalization operations and cuts, the latest cut before s is finalized reveals which records s successfully received and ordered: these records can be retrieved from any of the surviving storage servers in the shard. Records received but not incorporated in s's latest cut must be retransmitted by the originating clients to different shards. Corfu also responds to a storage server failure by finalizing its shard and adding a new one. During this process, however, all Corfu's shards are unavailable; in contrast, Scalog's non-faulty shards are unaffected (see §6.2).

Applications that require data locality may run application processes in storage servers (see §5.3). *Finalize & Add* would force those processes, if an entire shard is finalized, to migrate. Instead, Scalog supports two alternative options.

Remove & Replace (Requires at least $f+1$ storage servers per shard): As in vCorfu, Scalog can replace a failed storage server with a new one, which can then copy records from its shard's surviving storage servers. During this process, the affected shard is temporarily unavailable for writes (but continues to serve reads). This option suffers from a longer service recovery time [25].

Mask (Requires at least $2f+1$ storage servers per shard): At the cost of extra resources, this option ensures that, if no more than f of its storage servers fail, a shard will continue to process both reads and writes. This option also masks straggling storage servers. For long-term availability, new storage servers can be added to replace faulty ones; they can copy records from the shard's surviving servers.

	Replicas per Shard	Data Locality	Service Recovery Time
Finalize & Add	$f+1$	No	Short
Remove & Replace	$f+1$	Yes	Long
Mask	$2f+1$	Yes	Zero
Corfu	$f+1$	No	Short
vCorfu	$f+1$	Yes	Long

Table 2: Trade-offs of different approaches to handling storage server failures

All options guarantee linearizable semantics under crash failures, but they provide different trade-offs with respect to resource usage, data locality, and service recovery time after a failure. Table 2 summarizes these trade-offs and compares these options with failure recovery in Corfu and vCorfu.

4.4.3 Handling Ordering Layer Failures

Failures in the ordering layer can affect replicas running Scalog's ordering logic as well as aggregators. Replica failures are handled by Paxos; aggregator failures are handled by leveraging the statelessness of aggregators. A storage server or an aggregator that suspects its neighboring aggregator of having failed reports to the ordering layer, which responds by creating a new aggregator to replace the suspected one. A mistaken suspicion does not harm correctness, as both the new and the wrongly suspected aggregator correctly report local ordering information to their parent.

A distinguishing feature of Scalog is that Scalog suffers no net throughput loss because of ordering layer failures. Because of Scalog's approach to decoupling ordering from data replication, storage servers continue accepting client append requests and ordering records locally in their log segments, independent of the status of the ordering layer. Any temporary loss of throughput caused by an ordering layer failure is thus made up for as soon as the failure is recovered, when these locally ordered records are seamlessly inserted in the next cut issued by the repaired ordering layer. It does cause a spike in throughput because the repair interleaves all delayed records that are already replicated in one single cut. This is in contrast to sequencer-based logs where, after throughput halts because of a sequencer failure and reconfiguration [17, 21]), throughput goes back to normal instead of compensating for the loss of availability.

5 Applications

Applications can configure Scalog and customize sharding policies to satisfy their requirements. This section discusses typical applications that benefit from Scalog and demonstrates how to configure Scalog and set sharding policies.

5.1 The Online Marketplace

The online marketplace we used to motivate the Scalog design logs user activities (sellers listing products, buyers browsing and purchasing products, etc.) to Scalog for analytics and fault tolerance. To satisfy the requirements discussed in §2, we configured Scalog to use *Finalize & Add* to handle storage server failures and for a sharding policy we let each application process write to the nearest storage server.

If an application process writes at a rate that may overwhelm a single shard, it may select multiple shards to distribute the writes. Periodically, analytics jobs read Scalog, which may negatively affect the write rate; therefore, before performing analytics jobs, the online marketplace finalizes shards and adds new shards: the online marketplace writes to newly added shards, and analytics jobs read from finalized shards. This isolation makes sure analytics reads do not negatively affect online writes.

Using the API discussed in §2, the online marketplace calls `append` to log user activities. Periodically, analytics jobs use the `subscribe` API to extract data. When any of the system components fail, the online marketplace calls `subscribe` to replay the log and reproduce its state.

5.2 Scalog-Store

Modeled after Corfu-Store [21], Scalog-Store uses Scalog as its underlying storage. Scalog-Store configures Scalog to handle storage server failures using *Finalize & Add*, as it is the same as how Corfu handles failures. Like the online marketplace's sharding policy, the sharding policy is for an application process to select the nearest storage server.

Scalog-Store supports the same operations as Corfu-Store: atomic `multi-get`, `multi-put`, and `test-and-multi-put` (conditional `multi-put`). Scalog-Store uses a *mapping server* with an in-memory hash map that maps each key to a pair (l,s) containing a global sequence number l and a shard identifier s where the latest record containing the value of that key is stored.

To implement `multi-get`, which takes a set of keys as input, a client retrieves, in a single atomic request, the (l,s) pairs for the keys from the mapping server. The client then calls `readRecord`(l,s) for each pair to get each key's value.

To implement `multi-put`, a client first executes `appendToShard`$(\langle key, value\rangle)$ for each key to receive corresponding (l,s) pairs. Next, the client creates a *commit record* that contains the set of $(key,(l,s))$ records for each key and uses `appendToShard` to add the commit record to the log. (An optimization for single-key `multi-put` operations is only to log a commit record containing the key and value.) The client then forwards the commit record to the mapping server, which updates its hash map accordingly and responds. `multi-put` finishes on receipt of the response. Should the mapping server crash, a new server can re-read the log and rebuild a current hash map.

The implementation of `test-and-multi-put` is similar to that of `multi-put`, but adds a test condition to the commit record. Upon receiving the forwarded commit record, the mapping server evaluates the test condition to decide whether to commit the operation. If so, the mapping server processes the operation normally; otherwise, the mapping server processes the operation as a *no-op*. Finally, the mapping server returns the result to the client.

5.3 vScalog

Modeled after vCorfu [51], an object store based on Corfu, vScalog is an object store that runs on Scalog. The key difference between vScalog and vCorfu is how they guarantee data locality. vCorfu maintains a separate log, a so-called *materialized stream*, for each object, in addition to a global shared log. A client has to write an object update to the shared log for total order and to the materialized stream for data locality. vScalog, instead, leverages Scalog's sharding policy to map each object to one shard, effectively using each Scalog shard as a materialized stream. As a result, the single shared log guarantees both total order and data locality. vScalog can configure Scalog to handle storage server failures using either *Remove & Replace* or *Mask*; our implementation uses *Remove & Replace* because it is how vCorfu handles failures.

Compared with vCorfu, vScalog offers two main advantages. First, it is more robust: it can tolerate f failures in each shard, while vCorfu cannot handle a log replica and a stream replica failing simultaneously. Second, it offers higher read throughput: it lets clients read from all the replicas in a shard, while vCorfu's clients only read from stream replicas. A disadvantage is that vScalog requires all transactions, including those that will eventually abort, to be written to the log. The fundamental reason goes back to Scalog's persistence-first architecture, as the predicate on a `test-and-multi-put` operation may depend on the position of the corresponding record in the log, which Scalog decides after the record is replicated.

6 Evaluation

The goal of Scalog is to provide a scalable totally ordered shared log with seamless reconfiguration. In our assessment of Scalog, we ask the following questions:

- How do reconfigurations impact Scalog? (§6.1)
- How well does Scalog handle failures? (§6.2)
- How much write throughput can Scalog achieve and what is the latency of its write operations? (§6.3)
- How well do Scalog read operations perform in different settings? (§6.4)
- How do Scalog applications perform? (§6.5)

We have implemented a prototype of Scalog in golang [7], using Google protocol buffers [9] for communication. To tolerate f failures, the ordering layer runs Paxos with $2f+1$ replicas and each shard comprises $f+1$ storage servers; unless otherwise specified, we set $f=1$.

Some of our experiments use Corfu as a baseline for Scalog. To enable an "apples-to-apples" comparison, we implemented a prototype of Corfu in golang: it uses one server as a sequencer, $f+1$ servers for each storage shard, and Google protocol buffers for communication. Our Corfu implementation achieves higher throughput and lower latency than Corfu's open-source implementation [5]. To simplify comparison with published Corfu benchmarks, we fix the record size at 4KB.

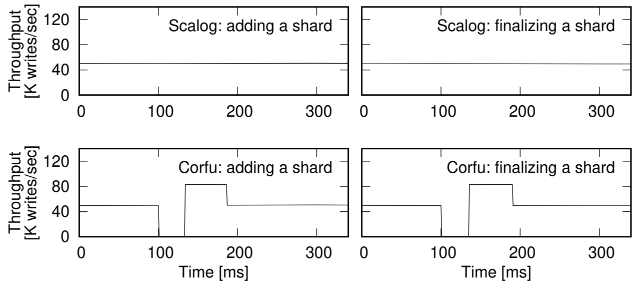


Figure 3: Throughput during reconfiguration

We run our experiments on 40 c220g1 servers in Cloudlab's Wisconsin datacenter. Each server has two Intel E5-2630 v3 8-core CPUs at 2.40GHz, 128GB ECC memory, a 480GB SSD, and a 10Gbps intra-datacenter network connection. Since exploring the limits of Scalog's write throughput requires many more than the 40 servers available to us, we resorted to simulation for results that report on larger configurations (specifically, those in Figure 5 in §6.3.2).

6.1 Reconfiguration

To evaluate how Scalog and Corfu perform when shards are added and finalized, we run both with six shards, each shard having two storage servers ($f=1$). We target 50K writes/sec, roughly half of the maximum throughput in this setting. We either add a shard or finalize a shard at $t=100$ms.

Figure 3 shows that Scalog's throughput is unaffected by adding or finalizing shards. When shards are added, clients can continue to use the original shards. Clients connected to shards are notified prior to finalization (we set the value of the configuration variable described in §4.4 to 10). During reconfiguration, throughput in Corfu ceases for roughly 30 ms because all storage servers must be notified before the new configuration can be used [21].

6.2 Failure Recovery

To evaluate how Scalog and Corfu perform under failure, we again deploy them with six shards, each with two storage servers, and use 50K writes/sec. To evaluate performance under aggregator failure, we add two aggregators to Scalog, each handling half of the shards. We measure throughput under four failure scenarios in Scalog: Paxos leader failure, Paxos follower failure, aggregator failure, and storage server failure, and under two failure scenarios in Corfu: sequencer failure and storage server failure. In each scenario, we intentionally kill one server at time $t=2$s and measure how throughput is affected. Figure 4 reports the results for the six failure scenarios:

Scalog's Paxos leader and Corfu's sequencer. Although records are temporarily unable to commit, Scalog's storage servers can continue receiving new records, which are com-

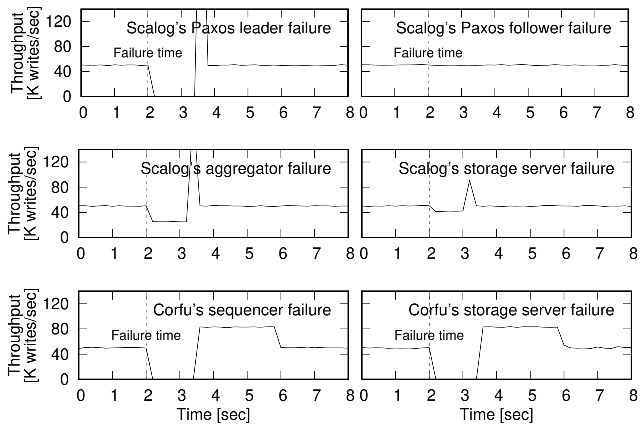


Figure 4: Throughput under different failure scenarios

mitted as soon as a new Paxos leader is elected. Hence, after a dip, throughput temporarily spikes to catch up, and total throughput is unaffected, although latency suffers until a new leader is elected. On the other hand, Corfu's clients compete for log positions when the sequencer is unavailable [21]. Heavy contention among clients causes Corfu's throughput to drop to nearly zero until a replacement sequencer joins [17]. Thereupon, Corfu runs at peak throughput until it catches up and stores all the delayed records; during this time, Corfu experiences higher latency.

Scalog's Paxos follower. No effect on throughput or latency.

Scalog's aggregator. Again, although the affected storage servers (in this case, half of all storage servers) are temporarily unable to commit new records, they can continue to receive them. Thus, the effects on throughput and latency are similar to those of a Paxos leader failure.

Scalog's and Corfu's storage servers. We compare Scalog's *Finalize & Add* with Corfu because they have the same trade-offs. In Scalog's *Finalize & Add*, the faulty server's shard is finalized. Throughput decreases temporarily until the failure is detected (relying, in our setting, on a one-second timeout) and all clients connected to the finalized shard are redirected to storage servers in other shards. Throughput is restored after slightly more than a second. In Corfu, the faulty storage server triggers a change in the mapping function; while this takes place, Corfu is unavailable [21]. Again, once the failure recovery completes, Corfu is saturated until all buffered records are stored.

6.3 Write Performance

We measure Scalog's write latency and throughput by running each client in a closed loop in which it sends a record and then awaits an acknowledgment. Latency measures the time difference between when a client sends the record and when it receives the acknowledgment. Throughput measures the number of write operations per second over all clients.

Corfu's peak throughput depends on the number of shards and the sequencer's throughput. Scalog's peak throughput depends on the number of shards and the configuration of the aggregators; in addition, it also depends on the length of the interleaving interval. By increasing the interleaving interval, Scalog can increase its throughput because a higher interleaving interval allows Scalog's ordering layer to manage larger numbers of shards and storage servers at the expense of higher latency. To compare fairly against Corfu, we run our evaluation with a fixed interleaving interval, set at 0.1ms to match Corfu's write latency. As we will see in §6.3.2, even with this short interval, Scalog already supports many more storage servers than we have resources to deploy.

6.3.1 System Configuration

In both systems, as the number of shards increases, ordering becomes a bottleneck. To properly configure each system to measure its peak throughput, we run microbenchmarks to determine, (1) the maximum throughput of a single shard (using $f+1$ storage servers) and (2) the maximum number of shards that their respective ordering layer can handle.

Throughput from one shard. A shard in Scalog peaks at 18.7K writes/sec; our implementation of Corfu, while outperforming previously reported figures for Corfu [21], reaches 13.9K writes/sec. The difference is due to how the two systems enforce total order at each storage server. In Scalog, where storage servers sequence records in the order in which they receive them, it is natural to write these records to disk sequentially. In contrast, records in Corfu are ordered by the sequencer, not by the storage servers. Records from different clients may reach storage servers out of order. Corfu storage servers first skip over missing records and later perform random writes to fix the log once those records are received.

The number of shards each system can support depends on the maximum load Scalog's aggregators can sustain and the maximum throughput of Corfu's sequencer.

Scalog's aggregators. We measure the number of shards and child aggregators that an aggregator can handle by having its neighboring servers (be they storage servers, the Paxos leader, or other aggregators) send synthetic messages. We find that each aggregator can handle either 24 storage servers (i.e., 12 shards in our $f=1$ setting) or 23 child aggregators, while the ordering layer can handle up to either 12 shards or 22 aggregators. We use these numbers to estimate the maximum number of shards that Scalog can support for a given number of aggregators.

Corfu's sequencer. We find that the sequencer of our Corfu implementation handles about 530K writes/sec, comparable to the optimized Corfu implementation used in Tango [22].

We want the throughput of both systems to scale linearly in the number of shards until ordering becomes the bottleneck. To avoid overloading the storage servers, we then configure each shard in Scalog and Corfu at 80% of their peak throughput, respectively, at 15.0K writes/sec and 11.1K writes/sec. To avoid overloading Scalog's Paxos leader and aggregators,

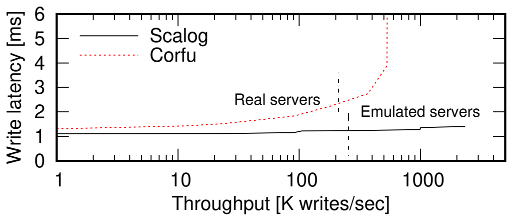


Figure 5: Latency vs throughput for Scalog and Corfu. The vertical dotted line separates results obtained with real servers from those obtained through emulation. For Scalog, we emulate storage servers, but not aggregators. Scalog's maximum throughput in this configuration is limited by the number of machines available to us.

we never assign to the ordering layer or to individual aggregators more than half of maximum load they can sustain (i.e., either six shards or 11 aggregators); if the load exceeds what the system's current configuration can handle under this policy, we add a new layer of aggregators. Thus, we configure these systems as follows:

Scalog. We add one shard for every 15.0K writes/sec of throughput. With up to six shards, we do not use aggregators. Between 7 and 66 shards, we use one layer of aggregation, with one aggregator for every six shards. With more than 66 shards, we use multiple layers of aggregators, where the ordering layer handles at most 11 aggregators, each aggregator handles at most 11 child aggregators, and each leaf aggregator handles at most six shards.

Corfu. We add one shard for every 11.1K writes/sec of throughput, until the sequencer becomes a bottleneck.

6.3.2 Write Scalability

We now proceed to determine how much load Scalog and Corfu can handle and, in particular, the throughput and latency that they achieve. Unfortunately, we only have access to 40 servers in CloudLab; in cases that require more servers, we emulate storage servers and their load. When communicating with the ordering layer, each (emulated) storage server reports to be receiving records at the same throughput and latency as a real storage server, though it is not receiving records from clients. This setup allows one physical machine to emulate hundreds of storage servers.

Let l_1 be the time elapsed at the client between submitting a record and learning that it is committed, and let l_2 be the time elapsed between submitting a report to the ordering layer and learning the corresponding cut. Both are measured using real storage servers. For our emulation, we use as latency the sum of (1) the time elapsed at the *emulated* storage server between submitting a report to the ordering layer and learning the corresponding cut and (2) $l_1 - l_2$.

Figure 5, which presents throughput/latency measurements as we increase the number of shards, shows that Scalog significantly outperforms Corfu's throughput while experiencing lower latency.

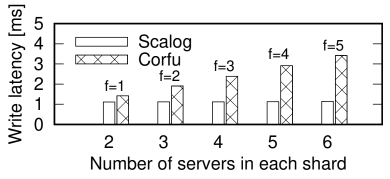


Figure 6: Write latency vs. shard size

Corfu's maximum throughput is limited by the sequencer at 530K writes/sec. Emulating only storage servers, but not aggregators, with our 40 machines, Scalog reaches 2.34M writes/sec, but is still far from being saturated. To explore the limits of the workload that can be handled by Scalog's ordering layer, we deployed Paxos with multiple layers of aggregators: we used physical servers for the Paxos replicas and the uppermost layer of aggregators, and emulated additional layers of aggregators as necessary against an emulated workload corresponding to a varying number of storage servers. We found that, before Paxos becomes a bottleneck, Scalog can handle up to 3,500 shards with three layers of aggregators, which translates to 52M writes/sec.[4] This throughput could be further increased by using a larger interleaving interval, trading latency for throughput.

Scalog's latency in Figure 5 grows slightly (by about 0.1 ms) whenever a new layer of aggregators is added, but remains lower than Corfu's. Based on our experiments with one and two layers of aggregators, we estimate the latency at 52M writes/sec to be around 1.6 ms (the client perceived latency is 1.3 ms when there are no aggregators, plus three layers of aggregators at about 0.1 ms per layer).

Corfu's latency is negatively impacted by two factors: first, Corfu replicates records across storage servers using client-driven chain replication [49] that writes to each server in sequence; second, since Corfu's clients may (and, in sufficiently long runs, likely will) write records to any storage server, the overhead paid by servers in managing client connections grows with the number of clients.

Finally, we investigate how write throughput and latency are affected by f, the number of failures that a shard tolerates. We find that throughput in both Scalog and Corfu is not significantly affected when varying f; thus, we focus our discussion on latency. Figure 6 shows that, for a single shard, client-perceived latency in Scalog is roughly constant, while in Corfu, latency increases linearly with f. The reason is, again, that Corfu replicates a record within a shard by writing sequentially to each of its storage servers, while Scalog allows a record to be replicated in parallel on multiple storage servers. Thus, as the number of storage servers in the shard increases to tolerate higher values of f, so does the latency gap between Scalog and Corfu.

[4]We use emulation to measure the maximum number of shards the ordering layer can handle. We are unable to assess other scaling issues (e.g., the network bottleneck), because we do not have access to a sufficiently large testing infrastructure.

6.4 Read Performance

Unlike writes, reads in Corfu and Scalog follow similar paths with identical performance. We therefore only focus on Scalog's read latency and throughput.

Using a single storage server s, we measure latency with a single client and measure throughput as a function of the number of clients. To evaluate the performance of sequential reads, we have a client call `subscribe`(l), where $l \leq l_{max}$, the maximum global sequence number the storage server has observed (see §4.2). We measure latency as the time between the `subscribe` call and the receipt of the first record; for throughput, we divide the number of records between $[l, l_{max}]$ in s by the time needed to receive them. To evaluate random reads, we have a client call `readRecord`(l,s) in a closed loop, where l is randomly generated such that $l \leq l_{max}$ and record l is stored in shard s.

Normally, the client library connects to all shards for `subscribe`(l) and chooses a random server in shard s for `readRecord`(l,s) (§4.2); instead, for these measurements we modified the client library so that it connects only to the storage server in s that is the focus of our evaluation.

When the client reads data that is still stored in the memory of the storage server, the throughput for both `subscribe` and `readRecord` is 280K records/sec (i.e., the limit of a storage server's network bandwidth) and the latency for both a `readRecord` request and for receiving the first record after a `subscribe` call is about 0.09 ms.

When the client reads data that is no longer in memory (as is often the case with finalized shards), latency and throughput are limited by the performance of storage server disks. With our hardware, `readRecord` achieves 4.5K records/sec throughput and 0.31 ms latency; as for `subscribe`, by reading sequentially and returning 256KB log chunks, it achieves 57K records/sec throughput with 1.21 ms latency to receive the first record; larger chunks improve throughput somewhat, but at the cost of significantly higher latency.

When a client reads from many storage servers concurrently (whether from one or multiple shards), throughput is limited by the client's network bandwidth, which is on average 280K records/sec in our evaluation.

6.5 Impact on Applications

We focus on the applications discussed in §5. Of these, the online marketplace uses Scalog to store user activities using `append` and reads the log using `subscribe`, so its performance is simply that of Scalog. The other two applications are more involved and deserve a more careful investigation.

6.5.1 Scalog-Store

We have implemented prototypes in golang using protocol buffers of both Scalog-Store and Corfu-Store based on our Scalog and Corfu implementations.

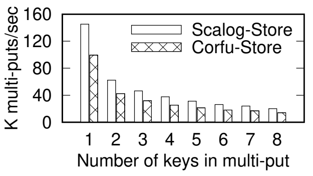


Figure 7: throughput of `multi-put` with 10 shards

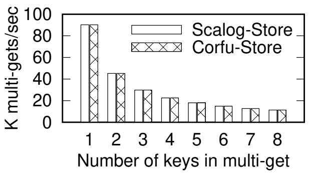


Figure 8: throughput of `multi-get` with 10 shards

In this experiment, both Scalog-Store and Corfu-Store run on 20 storage servers (10 shards). The keys are 64-bit integers, while values are 4088 bytes (creating 4KB records).

Figure 7 shows that Scalog-Store has higher `multi-put` throughput than Corfu-Store, because each storage server in Scalog has higher throughput (see §6.3.1). For both Scalog-Store and Corfu-Store, the throughput of `multi-put` operations is limited by the throughput of the log given the limited number of shards we have available. An exception is when Scalog-Store has 10 shards and one key in `multi-put`, when the bottleneck is the mapping server.

If we had many more shards but few keys in `multi-put` operations, then the mapping server would be the bottleneck for both Scalog-Store and Corfu-Store, and we would expect the `multi-put` throughput to be the same. However, if we increase the number of keys, we would expect Scalog-Store to eventually have higher throughput than Corfu-Store because the bottleneck will eventually shift to the log. This is because the throughput of the mapping server does not deteriorate much with the number of keys and the throughput that the log has to provide equals the number of keys times the throughput of the mapping server. For Corfu-Store, the shift happens when there are eight keys. Because Scalog provides superior throughput to Corfu, Scalog-Store can provide higher `multi-put` throughput when the number of keys is larger than eight.

To summarize, Scalog-Store achieves higher per-shard write throughput than Corfu-Store, because Scalog-Store uses fewer shards to achieve the same total throughput. When there are eight or more keys in each `multi-put` operation, Corfu reaches its maximum throughput and becomes a bottleneck while Scalog does not.

For both Scalog-Store and Corfu-Store, the throughput of `multi-get` operations (Figure 8) is limited by random read throughput of storage servers.

6.5.2 vScalog

Starting respectively from our Scalog and Corfu implementations, we prototyped vScalog and vCorfu in golang, using protocol buffers. We implemented each object as a key-value pair and ran each system as a key-value store.

We first measure the maximum write throughput of a single materialized stream, since it limits the maximum update rate of a single object. Our evaluation shows that one materialized stream of vScalog and vCorfu achieves 18.6K writes/sec and 13.6K writes/sec, respectively, which are roughly the same as the respective single shard throughputs of Scalog and Corfu shown in §6.3.1. The client perceived latencies for vScalog and vCorfu are 1.2ms and 1.5ms, respectively; vCorfu is slower because it writes to disks sequentially while vScalog writes to disks in parallel.

Next we measure the total throughput of vScalog and vCorfu. Our experiments show that, using the same number of shards, vScalog has roughly the same throughput as Scalog. Using the same number of stream replicas in vCorfu as the number of shards in Corfu, and given enough log replicas, vCorfu and Corfu also achieve approximately the same throughput. However, the single shard throughput of vCorfu's underlying shared log reduces to 9.3K writes/sec (due to the cost of writing a commit bit, matching the 40% penalty reported in [51]).

7 Limitations

Scalog's current prototype suffers from several limitations. Some seem to be relatively easy to address: for example, while Scalog allows applications to dynamically add and finalize shards, it does not provide automated policies to trigger such actions. Other limitations are common to storage systems that operate at large scale: as server failures become frequent, the steps needed for recovery may complicate the scheduling and allocation of resources. Others yet, however, appear to be more fundamental to Scalog's design. In particular, although Scalog offers unprecedented throughput at latency comparable to, or better than, prior shared log implementations, it is not well suited for applications that require ultra-low latency (such as high-speed trading), highly-predictable latency and throughput, or low tail latencies. The question of whether it is possible to drastically reduce latency while maintaining Scalog's throughput and ordering properties remains open. Finally, some issues are outside of Scalog's current scope: in particular, Scalog's design does not address security concerns.

8 Related Work

The shared log abstraction is, implicitly or explicitly, at the core of state machine replication protocols [46], and Scalog draws inspiration from several of them.

In Vertical Paxos [39], configurations can change from slot to slot, allowing for seamless reconfiguration similar to Scalog. Like Scalog, EPaxos [43] allows all replicas to accept client requests. However, EPaxos only builds a *partial* order consistent with specified dependencies among records; in addition, maintaining and checking dependencies creates a bottleneck. In networks that almost never reorder messages, NOPaxos [40] achieves very high throughput and low latency using a custom hardware switch to order records.

Mencius [42] and Derecho [23] partition log slots among multiple leaders. Essentially, each process creates a locally ordered log, which is then interleaved in round-robin order. Similarly, Calvin [48] dispatches to multiple sequencers transaction requests, which are compiled into batches. The batches are then interleaved in round-robin order to build a total order. Scalog generalizes this idea and allows for more flexibility in how the logs are interleaved, which is not sensitive to slow servers.

Kafka [36], a widely-used shared log system, uses sharding to scale and provides total order within each shard, but not across them. Pravega [15] provides a sharded log similar to Kafka and focuses on a rich set of reconfiguration operations that support scaling. FuzzyLog [41] builds a partially ordered log by tracking Lamport's happened-before relation [37] between records stored in different shards. DistributedLog [6] also supports sharding and provides a totally ordered log, but its single-writer-multiple-reader access model is not conducive to high write throughput.

To provide both total order and high throughput, it is necessary to separate ordering from data dissemination. Like Scalog, Corfu [21] separates ordering from data dissemination and relies on sharding. A function, maintained in ZooKeeper [33], maps sequence numbers to shards. A client first obtains a sequence number for a record from the Corfu sequencer, and forwards the record to the shard indicated by the mapping function. Each shard comprises a collection of replicas, each consisting of a *flash unit* (an SSD plus FPGA to implement a write-once block device), kept consistent using a variant of chain replication [49].

LogDevice [12] is similar to Corfu, but replaces the mapping function with a non-deterministic record placement strategy. By allowing clients to write to any shard, LogDevice achieves flexible data placement. However, all records still need to be ordered by a sequencer similar to DistributedLog's single writer [6], limiting throughput.

9 Conclusion

Inspired by crash-resistant storage systems, Scalog departs from previous implementations of the totally ordered shared log abstraction by making records persistent before determining their positions in the log. This simple but essential change of perspective lets Scalog scale out elastically and recover from failures quickly; allows applications to customize which storage servers should hold their records; and enables a new ordering protocol that, by interleaving the local orders built by each storage server as a side product of replicating records, achieves almost two order of magnitude higher throughput than the state-of-art shared log implementation.

Acknowledgments

We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their insightful comments. We also thank Matthew Burke, Natacha Crooks, Youer Pu, Chunzhi Su, Florian Suri-Payer, Cheng Wang, and Yunhao Zhang for their valuable feedback on earlier drafts of this paper. This work was supported by NIST F568386 and by NSF grants CSR-1409555 and CNS-1704742.

References

[1] Achieving high concurrency with ApsaraDB for RDS. https://www.alibabacloud.com/blog/achieving-high-concurrency-with-apsaradb-for-rds_594297.

[2] AlibabaMQ for Apache RocketMQ. https://www.alibabacloud.com/product/mq.

[3] Amazon Kinesis. https://aws.amazon.com/kinesis/.

[4] CloudLab. https://cloudlab.us.

[5] CorfuDB. https://github.com/CorfuDB/CorfuDB.

[6] DistributedLog. http://bookkeeper.apache.org/distributedlog/.

[7] The Go programming language. https://golang.org.

[8] Google Cloud Pub/Sub. https://cloud.google.com/pubsub/.

[9] Google protocol buffers. https://developers.google.com/protocol-buffers/.

[10] IBM MQ. https://www.ibm.com/products/mq.

[11] Kafka uses. https://kafka.apache.org/uses.

[12] LogDevice: distributed storage for sequential data. https://logdevice.io/.

[13] Microsoft Event Hubs. https://azure.microsoft.com/en-us/services/event-hubs/.

[14] Oracle Messaging Cloud Service. https://www.oracle.com/application-development/cloud-services/messaging/.

[15] Pravega. http://pravega.io.

[16] Taobao. https://www.taobao.com.

[17] Zlog: a high-performance distributed shared-log for Ceph. https://github.com/cruzdb/zlog.

[18] D. Abadi. Partitioned consensus and its impact on Spanner's latency. https://dbmsmusings.blogspot.com/2018/12/partitioned-consensus-and-its-impact-on.html, 2018.

[19] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[20] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Crash consistency: fsck and journaling. In *Operating Systems: Three Easy Pieces*. 2018.

[21] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: a shared log design for flash clusters. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[22] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: distributed data structures over a shared log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[23] J. Behrens, K. Birman, S. Jha, M. Milano, E. Tremel, E. Bagdasaryan, T. Gkountouvas, W. Song, and R. van Renesse. Derecho: group communication at the speed of light. Technical report, Cornell University, 2016.

[24] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems*, 1993.

[25] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: reducing the frequency of data loss in cloud storage. In *Preceedings of the 2013 USENIX Annual Technical Conference*, 2013.

[26] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR)*, 17(3), 1985.

[27] F. D. T. e Silva. Kafka: ordering guarantees. https://medium.com/@felipedutratine/kafka-ordering-guarantees-99320db8f87f, 2018.

[28] R. C. Fernandez, P. R. Pietzuch, J. Kreps, N. Narkhede, J. Rao, J. Koshy, D. Lin, C. Riccomini, and G. Wang. Liquid: unifying nearline and offline big data integration. In *CIDR*, 2015.

[29] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the International Conference on Foundations of Computations Theory, Lecture Notes in Computer Science*, volume 158. Springer, 1983.

[30] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of 1th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.

[31] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye. Building LinkedIn's real-time activity data pipeline. *IEEE Data Eng. Bull.*, 35(2), 2012.

[32] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 1990.

[33] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, 2010.

[34] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2000.

[35] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the VLDB Endowment*, 2000.

[36] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: a distributed messaging system for log processing. In *Proceedings of the NetDB*, 2011.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[38] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 1998.

[39] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC)*, 2009.

[40] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just say NO to Paxos overhead: replacing consensus with network ordering. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[41] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan. The FuzzyLog: a partially ordered shared log. In *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[42] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[43] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[44] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. In *Proceedings of the VLDB Endowment*, 2014.

[45] R. D. Schlichting and F. B. Schneider. Fail-Stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3), 1983.

[46] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4), 1990.

[47] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *Proceedings of the VLDB Endowment*, 2010.

[48] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[49] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[50] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein. Building a replicated logging system with Apache Kafka. In *Proceedings of the VLDB Endowment*, 2015.

[51] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi. vCorfu: A cloud-scale object store on a shared log. In *Proceedings of 14th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2017.

Frequency Configuration for Low-Power Wide-Area Networks in a Heartbeat

Akshay Gadre[1], Revathy Narayanan[1,2], Anh Luong[1], Anthony Rowe[1], Bob Iannucci[1], Swarun Kumar[1]
[1]*Carnegie Mellon University,* [2]*IIT Madras*

Abstract

Low-power Wide-Area Networks (LP-WANs) are seen as a leading candidate to network the Internet-of-Things at city-scale. Yet, the battery life and performance of LP-WAN devices varies greatly based on their operating frequency. In multipath-rich urban environments, received signal power varies rapidly with a low-power transmitter's frequency, impacting its transmission time, data rate and battery life. However, the low bandwidth of LP-WANs means that there are hundreds of operating frequencies to choose from. Among them, we show how choosing a select few of these frequencies($\leq$3.55%) effectively triples the battery life when compared to the rest for LP-WAN devices.

This paper presents Chime, a system enabling LP-WAN base stations to identify an optimal frequency of operation after the client sends one packet at one frequency. Chime achieves this by analyzing the wireless channels of this packet across many base stations to disentangle multipath and ascertain an optimal frequency that maximizes client battery life and minimizes interference. We implement Chime on a campus-scale test-bed and achieve a median gain of 3.4 dB in SINR leading to a median increase in battery life of 230% ($\sim$1.4-5.7 years), data rate by 3.3$\times$ and reduction in interference of 2.8$\times$ over commodity LP-WANs.

1 Introduction

Recent years have seen the emergence of Low-Power Wide-Area Networks (LP-WANs) as a promising technology to connect the Internet of Things. LP-WAN technologies (like LoRaWAN [2], SIGFOX [61], 3GPP's NB-IoT [35], LTE-M [5]) allow devices to send data at low data rate (few kbps) to base stations several miles away powered by batteries with targeted lifetimes of 5-10 years. However, recent studies [17, 19, 28] show a contrasting reality in dense urban deployments where LP-WAN clients deep inside buildings experience significantly lower battery lives ($\sim$1-2 yrs) owing to heavy signal attenuation. They further show that over 97% of the energy consumption in an LP-WAN client can be directly attributed to its radio front-end.

While many parameters influence the battery-drain from a client's radio front-end, the main parameter that it can control is its operating frequency. With the opening up of the TV whitespaces, narrowband LP-WAN clients have several hundreds of operating frequencies to choose from [16]. While there is rich work on spectrum sensing, particularly to avoid interference, in Wi-Fi [24] and LTE [38], LP-WANs differ in an important way: base stations span asymmetrically higher bandwidth compared to clients. This means that base stations can directly monitor multiple frequency bands and advise clients on frequencies with minimal interference. Yet, base stations are unaware of the precise signal power at which an LP-WAN client's signal will be received across frequencies. Our extensive experiments (Sec. 3) over a wide-area campus testbed show a promising opportunity in this respect: We show how, when set to a select few frequencies ($\leq$ 3.55% of all available frequencies), signals from an LP-WAN client are received at much higher signal power ($\sim$3-4 dB) at base stations. This increases client data-rate ($\sim$2-8 $\times$) and reduces their transmission time, effectively tripling their battery life[1] relative to the median frequency. Unfortunately, finding these optimal frequencies is challenging because they correlate poorly by interpolating measurements along space, time or frequency of operation. Further, simply sifting through even a few frequencies (e.g. as with Wi-Fi [11]) in the hope of finding the optimal ones would itself drain the battery inordinately.

This paper presents Chime, a solution that explores the feasibility of offloading the LP-WAN client frequency configuration problem to the more well-equipped LP-WAN base stations. Chime considers static clients (e.g. sensors) in urban environments whose multipath characteristics, while complex, change relatively slowly over time. Chime uses the fact that while a single base station cannot ascertain the complex multipath, multiple spatially-distributed LP-WAN base stations

[1]Battery Life estimates derived from prior energy models (see Sec.3)

can collaboratively identify an optimal operating frequency for a client based on a single association packet it transmits when it wakes up, regardless of its initial operating frequency. Such an association packet is a standard feature of many LP-WAN protocols [51] posing minimal power overhead for the LP-WAN client. Chime achieves this by building a novel system that uses the wireless channel-state information of this packet at one frequency received across multiple base stations to disentangle the multipath and predict the long-term battery drain for the different operating frequencies. Further, Chime also predicts the extent of unwanted interference the client produces at base stations across different frequencies. Thus, Chime passively infers an optimal client operating frequency, without prior calibration of the environment or known client location.

Chime exploits the recent trend of massive and unplanned deployment of LP-WAN base stations [32]. For instance, LoRaWAN base stations are proposed to be deployed in Comcast MachineQ set-top boxes [32, 36], meaning that many LP-WAN base stations will likely be single-antenna and often deployed indoors. Chime proposes a novel algorithm (see Sec. 5) to synchronize these multiple single or multi-antenna LP-WAN base stations to emulate a large city-scale distributed antenna array. In particular, Chime builds on past work in the cellular context (e.g. R2-F2 [50]) that separates signal paths using multi-antenna arrays while dealing with new challenges pertaining to distributed, irregular arrays of antennas and low-power user devices. Chime first models the received signals from a client across synchronized base stations to disentangle the different paths that the signals may traverse as they reflect off different objects. These signals combine to reinforce or cancel each other leading to varying signal power across the operating frequencies of the client. Chime then estimates how these separated signal components recombine at different client frequencies to find the one maximizing battery-life and throughput, while minimizing interference.

A key challenge in estimating the multipath in urban environments is receiving time-synchronized phase measurements across multiple LP-WAN base stations to emulate a distributed MIMO array. While recent work has successfully demonstrated distributed MIMO for WiFi [22] and cellular networks [40], LP-WAN packets last over $10\times$ longer and therefore require much more accurate and long-lasting phase synchronization. Further, low-power devices experience large hardware imperfections meaning that the phase of the wireless channel varies drastically even within one packet. Hence, any phase measurements made over time across base stations would simply appear unsynchronized and random. Chime overcomes this challenge by never measuring the phase of a low-power client in isolation, instead always measuring it relative to a high-power master base station whose signal propagation characteristics we know *a priori*. We design this master base station's signal so that it can be measured at exactly the same time and frequency as the low-power client, without significantly interfering with it. Sec. 5 describes this novel algorithm that directly compensates for phase drifts over time of the low-power client relative to a reference signal due to hardware imperfections.

Next, Chime must use the synchronized phase measurements of a client's association packet to infer how the signal propagates through environment. However, inferring all paths of the signals using measurements from a few single-antenna base stations [53] that are geographically separated is challenging. Chime exploits the fact that though wireless signals in urban wide-area networks traverse diverse paths to base stations at different locations, they often share a very small number of common large reflectors (e.g. buildings, trees, big vehicles, etc.). Our approach aims to discover these dominant reflectors in the environment using the small number of wireless channel measurements and model the signal propagation (Sec. 6), while accounting for variations in the size, shape and orientation of these reflectors. Chime recombines signals in these dominant paths across frequencies to accurately predict an optimal frequency of operation for improved throughput and lower interference (Sec. 7).

Limitations and Scope: We emphasize that Chime: (1) Considers static LP-WAN clients (e.g. sensors, metering devices); (2) Models macroscopic environmental changes but neglects fleeting reflectors (trade-offs discussed in Sec. 6.2) (3) Assumes LP-WAN clients send an association packet to base station upon waking up. Yet, Chime remains broadly applicable to most sensor networking deployments.

Evaluation and Results: We deploy Chime using LoRa as the low power technology and Semtech SX1276 chips as the client RF transceivers. Our base-stations are USRP N210s deployed on six buildings in a 0.7 km $\times$ 0.5 km area surrounding a university campus. Our results show that:

- Chime provides a net increase in battery-life of 1.4-5.7 years (230%) achieving at an average 79% of the optimum.
- Chime can increase network throughput by $3.3\times$ compared to commodity LoRa.
- Chime can reduce interference from LP-WAN clients at base stations by 2.1 dB, by predicting the weakest frequency.

Contributions: Our specific contributions include:

- A wide-area motivation study that demonstrates the inability of spectral, temporal and spatial interpolation for identifying an optimal operating frequency of an LP-WAN client.
- A novel solution for collaboratively identifying an optimal operating frequency of an LP-WAN client at the base stations using only one transmitted packet from the client.
- A system that demonstrates significant increase in battery life and throughput by identifying an optimal frequency of operation for LP-WAN radios while accounting for multipath, interference and noise.
- A wide-area deployment across a university campus showing 1.4-5.7 years of increased battery-life for LP-WAN clients.

2 Related Work

Related work can be broadly categorized as follows:

Low-Power Wide-Area Networks: Recent years have witnessed much interest in LP-WANs on both cellular (LTE-M [5] and NB-IoT [35]) and unlicensed spectrum (Semtech's LoRa [2,27,44] and SigFox [39,61]), with some proposals extending to the TV whitespaces [20]. Recent work on LP-WAN has explored interference management [19,23], developing battery-free solutions [30,45] and system deployments on the whitespaces [48] to name a few. Chime complements this past work by considering rapid frequency configuration, a problem crucial for tackling with rapidly changing channel quality in urban spaces and improving battery-life.

Spectrum sensing: Cognitive radio and spectrum sensing solutions are primarily aimed at identifying vacant frequency bands to minimize interference with other users [56]. Many of these solutions rely on long-term statistics of channel occupancy and signal power using temporal [9,15,29,60] and spatial correlation [12,14,46,59] to make predictions. More recent work attempts to minimize feedback by relying on sparse recovery techniques such as compressed sensing [34,43,55] or eigen-value based methods [8,57].

For LP-WANs, channel occupancy, interference and noise can be directly inferred by the base stations because they span much larger bandwidth [10] compared to clients. Further, past work on spectrum sensing does not focus on predicting received signal power at the base station from a client across frequencies. This is precisely Chime's goal based on measurements at one frequency from a single client radio.

Optimal radio configuration: Perhaps the solutions most closely related to this paper are systems in the Wi-Fi [41] and cellular context [50]. CSpy [41] exploits the properties of OFDM wide-band transmissions from Wi-Fi client on one Wi-Fi frequency band to accurately model wireless channels at other Wi-Fi frequency bands. R2-F2 [50] predicts both channel magnitude and phase of LTE cellular signals based on measurements in one frequency, exploiting the properties of OFDM and large multi-antenna base stations.

In contrast the LP-WAN context brings three unique challenges to the problem of finding an optimal operating frequency. First, there are too many frequencies to choose from across whitespaces ($\sim$800MHz of bandwidth). For example, just running through all of them will consume about 6% of the client's battery life[2]. Second, these radio configurations demonstrate extremely poor correlation across frequency, time and space, ruling out statistical techniques to estimate the optimal frequency of operation (see large-scale study in Sec. 3). Finally, the vast majority of LP-WAN base stations are single-antenna [36] and often deployed indoors, ruling out past work that exploits bulky and expensive multi-antenna array infrastructure [7,50]. Indeed, while Chime builds on R2-F2 [50], analyzing multipath across distributed single-antenna base stations for frequency configuration in the LP-WAN context is its key contribution.

[2]Available Battery Energy: 2900mAh; 125kHz channels in 800MHz: 6400; Energy of a typical LoRa packet: 100 mAs ; Battery spent = 6.13%

3 Motivation - Empirical Study

To motivate the battery-saving opportunities in finding an optimal frequency configuration and the core-challenges in finding it, we present our findings from a detailed empirical study. We focus on a simple question: "Can an optimal frequency of operation of an LP-WAN client be found by exploiting prior measurements made over time, frequency or space?". We deploy 20 LoRaWAN clients at multiple locations periodically sending packets across a month iterating over 160 frequency configurations in an outdoor campus-scale testbed (see Sec. 9 for a detailed description of our testbed). Each client was static and placed in a weather-proof case in indoor and outdoor locations with signal power measured from a single base station. While we do not consider mobile clients, we consider varying outdoor environments over time to measure channel quality and estimated battery life for each frequency.

Estimating battery life: Prior studies have shown that the RF front-end is responsible for most of the battery consumption of a LoRaWAN device [17]. We use these prior LoRaWAN battery models [17] to estimate the energy consumed per packet at different datarates. We then use operational characteristics for the Semtech SX1276 transceiver [4] to map the signal-to-interference plus noise ratio (SINR) to the appropriate datarates. They show that improving signal strength from a LoRaWAN client can reduce the transmission time, thus increasing battery-life. The reason the battery life increases so much with a few dB improvement in SINR is that, unlike WiFi, every better data rate in LP-WANs halves the packet transmission time [4]. Thus, across the SINR thresholds of these data rates, your client battery life doubles, quadruples and so on. Our results show high variation in the RSSI of a LoRa client at base stations across time, frequency and space.

Correlation across time: Upon investigating the data across clients to the base station, we discover that most frequencies change in signal strength even across a few minutes (Fig. 2). Our results (Fig. 3) show that using historical measurements over different time spans on a set of frequencies to predict the optimal one (via polynomial interpolation) achieves 38.27% of the optimum at best. Our detailed study of urban multipath in Sec. 10.2 shows that this stems from gradual aggregate change in reflectors in the environment at these timescales.

Correlation across Frequency: Our results reveal that the optimal frequency-of-operation is extremely difficult to stumble upon with a random guess or even predict using a modest amount of frequency hopping. As shown in Fig. 1, 50% of all operating frequencies provide just 27.58% of optimum battery life while 90% of them still provide only 67.09% of the optimum. Indeed, only 1.58% of the operating frequencies are

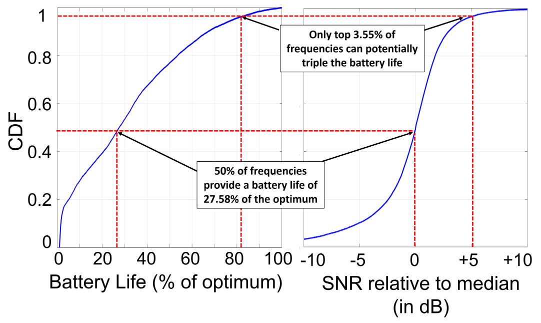


Figure 1: **Percentile of battery life**: few frequencies have good SINRs providing battery lives close to the optimum

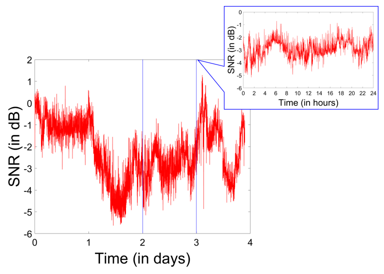


Figure 2: **Channel Variance**: Channel quality varies dynamically across days and even minutes

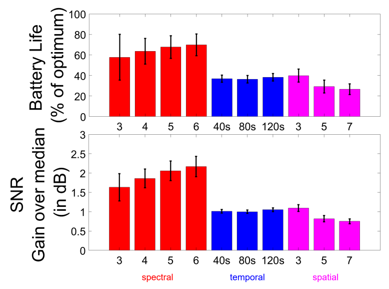


Figure 3: **Interpolation**: percentile of battery life of the optimum frequency from interpolation

at 90% of the optimum while only 3.55% triple the median battery life. Further, sampling several frequencies in hope of finding the top 3.55% would itself incur battery-drain, zeroing out the benefits. We observe that even adjacent transmission frequencies perceive a difference of about 20 dB of signal strength which, in outdoor environments, is enough to make a LoRa device undetectable. We further evaluate whether polynomial interpolation from sampling a limited number of frequencies sufficiently improves battery life and observe (Fig. 3) that it achieves at best 70.07% of the optimum.

Correlation across Space: We evaluate whether measurements from neighboring clients can be leveraged to find an optimal frequency of operation for a client. We consider various number of clients placed in a linear array spaced at 15 cm and predict an optimal frequency of the client in the middle via polynomial interpolation. As shown in Fig. 3, this achieves at best 39.90% of the optimum battery life.

4 Overview of Chime

This section provides an overview of Chime's approach and challenges. Chime's primary goal is to accurately measure an optimal operating frequency for an LP-WAN client by making it transmit only one packet on one frequency band. It primarily aims to predict the received signal power of the client across all frequencies at base stations. Since base stations span a wide bandwidth, they can readily measure channel occupancy and noise levels across frequencies, leaving received signal power from a client as the primary unknown.

Chime's system architecture is designed as follows: Upon waking up and for signal association, each LP-WAN device transmits a beacon packet on its arbitrarily chosen initial frequency of operation (a standard feature of common LP-WAN protocols). Chime then processes the received signals from this packet across the base stations at the cloud via a wired backhaul to infer an optimal frequency of operation. Note that since the powered base stations and the cloud perform all computation, this does not impact client battery life. The nearest base station then reports the estimated frequency to the client in its acknowledgment of the beacon.

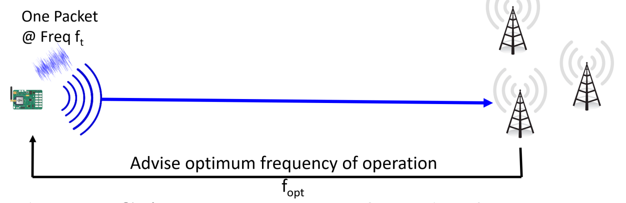


Figure 4: **Chime**: Frequency configuration for LP-WANs

Assumptions: While Chime does not consider mobile clients, we do consider dynamic outdoor environments. While Chime does not model fleeting reflectors in environment, it models long-term changes in multipath as it re-analyzes the current multipath based on transmissions from the client beacon and the master base station.

The rest of this paper describes three challenges in achieving the above design: (1) ***Synchronizing Distributed Base Stations:*** Chime first develops a synchronization system that allows multiple base stations to coordinate. In doing so, it eliminates the time-varying and long-lasting phase errors due to hardware impediments, such as frequency, timing and phase offsets of low-cost and low-power wireless hardware (see Sec. 5). (2) ***Disentangling Signal Paths:*** Next, Chime analyzes the root cause of why signal power from the client would vary across frequencies in the first place – wireless multipath. Specifically, signals from the client traverse multiple paths as they reflect off buildings, trees and other objects before reaching the base stations. Signals along these paths can reinforce each other or cancel each other, depending on the frequency of operation. At the cloud, Chime combines measurements from the distributed array of base stations to decouple the different paths the signal traversed from the client, even if the geometry of these base stations is arbitrary and the environment is multipath-rich (see Sec. 6). (3) ***Estimating Optimal Frequency:*** Chime then recombines the signal components at all possible operating frequencies to determine their expected signal power across base stations. Chime can then use this information, along with the known interference and ambient noise at these frequencies perceived at base stations to determine the best frequency-of-operation (see Sec. 7).

5 Synchronizing Base Stations

In this section, we describe our approach to synchronize transmissions from the LP-WAN client between spatially distributed base stations. Recall that Chime relies on synchronized phase measured across different base stations from a single client device to extract signal multipath. However, these phase measurements experience time-varying errors owing to the hardware imperfections of LP-WAN radios. Four distinct hardware impediments contribute to these phase errors: (1) Carrier Frequency offset (CFO): occurs due to subtle differences between the carrier frequency that any two radios operate on; (2) Sampling Frequency offset (SFO): occurs due to small differences between the sampling rate of the two radios; (3) Detection Delay: is produced because the packet from the client is detected with different delays across base stations; (4) Phase Lock Loop (PLL): produces an arbitrary constant phase offset at each base station's received signal, every time it tunes to a frequency. Chime's synchronization algorithm seeks to process these wireless channels across base stations to eliminate these phase errors.

Let the measured channel between the client and the base station be denoted by $\tilde{h}_{C\to B_1}$ whose phase is $\tilde{\theta}_{C\to B_1}$. Mathematically, we can write the phase of the measured wireless channel $\tilde{\theta}_{C\to B_1}$ at time t as a function of the phase of the true channel $\theta_{C\to B_1}$ between them, as well as various phase errors. Let us define the following hardware impediments: (1) Carrier Frequency Offset (CFO): $f_C - f_{B_1}$ as the difference in carrier frequency between the client and base station. (2) Detection Delay and Sampling Frequency Offset (SFO): $t_C - t_{B_1}$ denote the effective offset in time owing to detection delay at the base station and sampling frequency offset. (3) Phase offset from the PLL: $\phi_C - \phi_{B_1}$ the phase error owing to the PLL of the client and base station locking to different values each time these radios start receiving at a center frequency. The phase of the channel at time t is:

$$\begin{aligned}\tilde{\theta}_{C\to B_1} = \theta_{C\to B_1} - (2\pi(f_C - f_{B_1})t \\ + 2\pi f_C(t_C - t_{B_1}) + (\phi_C - \phi_{B_1}))\end{aligned} \tag{1}$$

The rest of this section describes our approach to eliminate each of the above errors across base stations.

5.1 Eliminating Phase Errors

To eliminate phase errors in Eqn. 1, Chime leverages multiple base stations. Specifically, we recall that a client's transmission at time t can be recorded by multiple base stations, which can measure the corresponding wireless channels. Chime eliminates hardware impediments by exploiting the common phase shifts they induce to these channels.

Mathematically, Chime estimates the wireless channel at a second base station B_2 from the same client at the same time t. This wireless channel is written as:

$$\begin{aligned}\tilde{\theta}_{C\to B_2} = \theta_{C\to B_2} - (2\pi(f_C - f_{B_2})t \\ + 2\pi f(t_C - t_{B_2}) + (\phi_C - \phi_{B_2}))\end{aligned} \tag{2}$$

By subtracting Eqn. 1 and Eqn. 2 above, we get:

$$\begin{aligned}\tilde{\theta}_{C\to B_2} - \tilde{\theta}_{C\to B_1} = \theta_{C\to B_2} - \theta_{C\to B_1} \\ +2\pi(f_{B_2} - f_{B_1})t + 2\pi f(t_{B_2} - t_{B_1}) + (\phi_{B_2} - \phi_{B_1})\end{aligned} \tag{3}$$

Note that the above difference in phases is independent of hardware impediments owing to the client, i.e. its center frequency f_C, time-delay t_C or initial phase ϕ_C. However, as it is dependent on the impediments of the two base stations, Chime still needs to estimate the phase errors due to hardware differences between pairs of spatially distributed base stations.

To estimate these phase differences, Chime relies on a master base station (B_M, one of the base stations) at a known location. The master sends a signal at the same time t and frequency f_C as the client (we address the challenges in achieving this without causing collisions in Sec. 5.2). We then measure the difference in phase at the two base stations of the channel from the master base station:

$$\begin{aligned}\tilde{\theta}_{B_M\to B_2} - \tilde{\theta}_{B_M\to B_1} = \theta_{B_M\to B_2} - \theta_{B_M\to B_1} \\ +2\pi(f_{B_2} - f_{B_1})t + 2\pi f(t_{B_2} - t_{B_1}) + (\phi_{B_2} - \phi_{B_1})\end{aligned} \tag{4}$$

Notice that Eqn. 3 and Eqn. 4 have the same effect of hardware impediments on their right-hand side. By subtracting these two phase values, we obtain a quantity independent of hardware offsets:

$$\begin{aligned}\tilde{\theta}_{C\to B_2} - \tilde{\theta}_{C\to B_1} - \tilde{\theta}_{B_M\to B_2} + \tilde{\theta}_{B_M\to B_1} \\ = \theta_{C\to B_2} - \theta_{C\to B_1} - \theta_{B_M\to B_2} + \theta_{B_M\to B_1}\end{aligned} \tag{5}$$

The above quantity is independent of hardware offsets of the client and base stations and therefore directly captures the multiple signal paths along which the signal traverses. Assuming the channel between master base station and other base stations can be computed (described in Sec. 5.2) at the same time and frequency as the client, the term $\theta_{B_M\to B_1} - \theta_{B_M\to B_2}$ can be compensated for. Chime therefore estimates the following product of channels h_{12}^{conj} – a complex number we call the *offset-free channel* whose phase value is exactly $\theta_{C\to B_2} - \theta_{C\to B_1}$ – a function purely of the client and base stations (note: $(.)^*$ is the complex conjugate).

$$h_{12}^{\text{conj}} = \frac{\tilde{h}_{C\to B_2}(\tilde{h}_{C\to B_1})^* \tilde{h}_{B_M\to B_1} h_{B_M\to B_2}}{\tilde{h}_{B_M\to B_2} h_{B_M\to B_1}} \tag{6}$$

Chime can then use this offset-free channel, which is free of all time-varying phase offsets, to disentangle signal paths from the client, without being impacted by hardware impediments (Sec. 6). Note that while the phase of a single offset-free channel is ambiguous due wrapping of phase over 2π, we combine the information across multiple such channels to estimate the multipath. This approach resembles that of many phase-based localization systems [26, 49, 54].

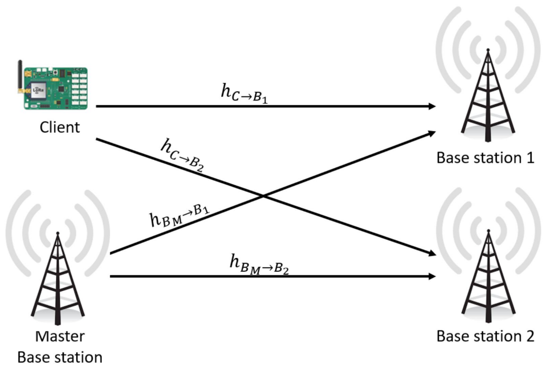


Figure 5: **Wireless channels** between client and base stations

5.2 Removing offsets between base stations

To obtain offset-free channels as in Eqn. 6 above, the base stations need to measure channels from the master base station at the same time and frequency as the client that is being tracked . However, doing so would result in collision between the master base station packet and client packet, causing neither of their packets to be decoded. As a result, one needs to carefully design transmissions of master base station to avoid collision with the client transmissions.

A naive approach would be to transmit the master's signal a short time interval prior to every client's transmission. By picking an extremely short interval between the reference and client, one can neglect the additional phase drift that may accumulate. While this approach is commonly used in distributed MIMO in Wi-Fi [33] and cellular [42], it does not apply to LP-WANs. This is because LP-WAN packets span hundreds of milliseconds [2, 13, 21]. Such long packets cause phase measurements to drift significantly within a packet rendering *a priori* synchronization futile. Thus, Chime needs a mechanism to estimate phase measurements at the same exact time and frequency from both reference and client by analyzing their packets transmitted concurrently.

Chime circumvents this challenge by designating one of the base stations to transmit a concurrent signal on an adjacent frequency band relative to the client. This **master base station** transmits its signal at the same time as the client, sending a known sequence in parallel with its transmission. The base stations can thus estimate the wireless channels of both the master and client transmissions at the same time, albeit across adjacent frequency bands. Chime then extrapolates the wireless channels of both the master and client to estimate its phase value at the guard band between them. While prior works [42, 58] have used beacon-based synchronization mechanisms, Chime uses piece-wise cubic spline extrapolation of both the magnitude and phase of the wireless channels across these bands for the master base station and client to estimate the magnitude and phase at the guard band in-between. Given that these estimates occur at the same time and frequency (i.e. the guard band) across both the master and client, we can now use them in Eqn. 6 to accurately synchronize base stations and eliminate the effect of hardware imperfections.

When does the master base station transmit? To facilitate the master base station to decode the preamble and transmit simultaneously on an adjacent channel, Chime ensures that the association packet's preamble is sufficiently long to accommodate this. An alternative option in the cellular context (e.g. NB-IoT) is to allocate dedicated spectrum for the base station alongside the client's association packet.

Why does interpolation work? Interpolation across frequency to estimate the channel seems to have inherent contradiction with our motivation results in Section 3. However, it is a well known fact that outdoor channels, have a coherence bandwidth of about 250-500 KHz. Thus, while the channel demonstrates frequency-selective fading over large bandwidths, the narrowband channel over 125 KHz is relatively flat [37]. Thus, interpolation of client and master base station channel will give a reasonable estimate of their channel at the guard band. Note that since the base stations are high powered agents, they can indeed transmit constantly and will have significantly larger transmit power than the clients. Thus, with a dense enough deployment of base stations (expected for LP-WANs [3, 6, 36]), the signal of the master base station will be received at other base stations.

6 Separating Signal Paths

Given the wireless offset-free channels of the form h_{jk}^{conj} from a client to a base station pair (j,k), we next seek to separate the set of signal paths that signals traverse from the clients to the n base stations. The key challenge in doing so is to decouple the large number of signal paths using channel measurements from a small number of base stations. Fortunately, our results in Sec. 10 as well as extensive past literature [50] in outdoor urban wireless networks demonstrate that wireless channels tend to have small number of dominant paths. As a result, Chime exploits this sparsity to identify the dominant signal paths using only a small number of available base stations. While there have been solutions proposed for WiFi [41] and cellular networks [50], these techniques either model certain behavior of signals in indoor environment or require heavy infrastructure such as an array of antennas unavailable at the base station. Furthermore, LP-WAN base stations are arranged irregularly, making it challenging to employ traditional antenna array algorithms.

6.1 Irregular Distributed Arrays

Chime separates multiple signal paths by actively modeling wireless signal characteristics of a distributed array of base stations with an irregular, but known geometry. To do so, Chime uses a maximum-likelihood [18] approach to identify the best propagation characteristics that fit the observed channels. In particular, given that only a small number of signal

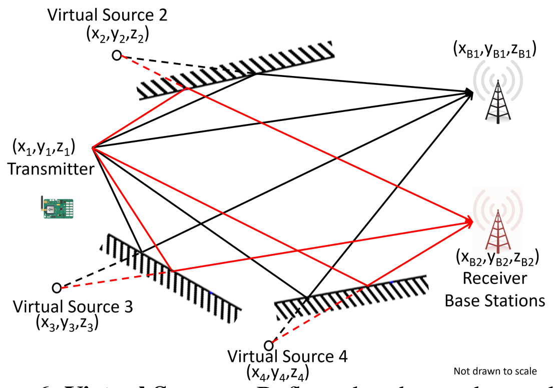


Figure 6: **Virtual Sources:** Reflected paths can be modeled as virtual sources that are mirror images of the transmitter

paths dominate (Sec. 10.2), Chime iterates over a set of m virtual source coordinates (x_p, y_p, z_p) for $p = 1, \dots, m$, which denote candidate locations for the client as well as one virtual source for each dominant path from a reflecting surface. As shown in Fig. 6, these virtual sources are simply the mirror image of the source about any reflecting surface. One can then compute the distances from these virtual sources to each base station (whose coordinates are known) to compute the total path length experienced by each reflected signal component. Chime then uses this information to compute the optimal attenuations and phase-shifts for each signal path that fit the observed channels and the given geometry of virtual sources. It then identifies and outputs the set of virtual sources, attenuations and phase shifts that best-fit the observed channels. Key to Chime's algorithm is an approach that both carefully chooses the number of paths m and efficiently searches over the space of virtual source coordinates.

Problem Formulation: Mathematically, Chime's algorithm begins by iterating over a set of candidate locations for the virtual sources corresponding to a client. For ease of exposition, we make two simplifying assumptions which we will relax later in this section: (1) Dominant reflectors are large, therefore shared by all base stations; (2) Dominant reflectors are planar and infinite. We further only consider single-bounce reflectors and assume multi-bounce reflected paths can be broken down into equivalent single-bounce reflectors. Let us assume for the moment that there are m such sources with known coordinates: (x_p, y_p, z_p) for $p = 1, \dots, m$. Let us denote $\mathcal{B}_q$ to be the coordinates of the n base stations. Consider a signal along path p traversing a distance of $d_{pj} = ||(x_p, y_p, z_p) - \mathcal{B}_j||$ to base station j from its virtual source (x_p, y_p, z_p). Then the phase of the channel from this source is of the form $-2\pi\frac{d_{pj}}{\lambda}$ and magnitude $\frac{1}{d_{pj}}$, where λ is the signal wavelength [47]. Let h_{jk}^{conj} denote the offset-free channels (see Sec. 5) received by each pair of base stations (i, j). Recall that h_{jk}^{conj} contains the product of channels to two base stations $h_{C\to B_j} h^*_{C\to B_k}$ so the phases of each pair of signal paths subtract and their magnitudes multiply. Hence, h_{jk}^{conj} is a weighted sum of complex numbers whose phase is of the form $-2\pi\frac{d_{pj}-d_{qk}}{\lambda}$ and magnitude is of the form $\frac{1}{d_{pj}d_{qk}}$, whose weights are unknown.

At this point, we formulate the following minimization problem that attempts to find the complex weights, $\alpha_{p,q}$, based on how well they fit the observed channels:

$$\min_{\{\alpha_{p,q}\}} \varepsilon$$

$$\left|\left| \left[h_{jk}^{\text{conj}}\right]_{1\times n^2} - [\alpha_{p,q}]_{1\times m^2} E_{m^2\times n^2} \right|\right| \le \varepsilon$$

$$E_{m^2\times n^2} = \left[\frac{1}{d_{pj}d_{qk}} e^{-i2\pi\frac{d_{pj}-d_{qk}}{\lambda}}\right]_{p,q=1,\dots,m;\ j,k=1,\dots,n}$$

where $i = \sqrt{-1}$. Given d_{pj}'s and d_{qk}'s, the above optimization problem can be solved in closed-form using a least-squares fit as (note: $(.)^{\text{pinv}}$ is pseudo-inverse.):

$$\alpha^{\text{est}} = \left[h_{jk}^{\text{conj}}\right]_{1\times n^2} E^{\text{pinv}}_{m^2\times n^2} \tag{7}$$

At this point, we can estimate the goodness-of-fit of the assumed coordinates of the virtual sources corresponding to the client $\{(x_p, y_p, z_p)\}_{p=1,\dots m}$ based on how well the estimated channels agree with the observed channels. We define the goodness-of-fit of virtual source coordinates $\{(x_p, y_p, z_p)\}_{p=1,\dots m}$ as:

$$G(\{(x_p, y_p, z_p)\}_{p=1,\dots m}) = 1/\left|\left|\left[h_{jk}^{\text{conj}}\right] - \alpha^{\text{est}} E\right|\right|$$

Thus, our problem of disentangling the multipath reduces to finding the coordinates of virtual sources in a given geographical domain $\mathcal{D}$, $C^{\text{opt}} = \{(x_p^{\text{opt}}, y_p^{\text{opt}}, z_p^{\text{opt}})\}_{p=1,\dots,m}$ as:

$$C^{\text{opt}} = \underset{\{(x_p,y_p,z_p)\}_{p=1,\dots,m}\in\mathcal{D}\}}{\arg\max} G(\{(x_p, y_p, z_p)\}_{p=1,\dots,m})$$

Run-time Optimization: Running the above optimization through an exhaustive grid search is prohibitive. Instead, Chime solves it numerically using a stochastic gradient descent algorithm [25] that begins optimization at a few of initial points (e.g. a coarse grid) in parallel. We then perform a finer numerical gradient-based search at these points and report the coordinates for which we obtain the global maximum of goodness-of-fit. Also, prior information about the topography of the deployment space, known reflectors, and location of the transmitter, while not necessary, can speed up the search process. Upon optimization, Chime can fully characterize the m dominant taps by the virtual source coordinates C^{opt} and corresponding phase shifts: $\alpha^{\text{opt}} = \left[h_{jk}^{\text{conj}}\right] E^{\text{pinv}}$.

6.2 Designing Optimization Parameters

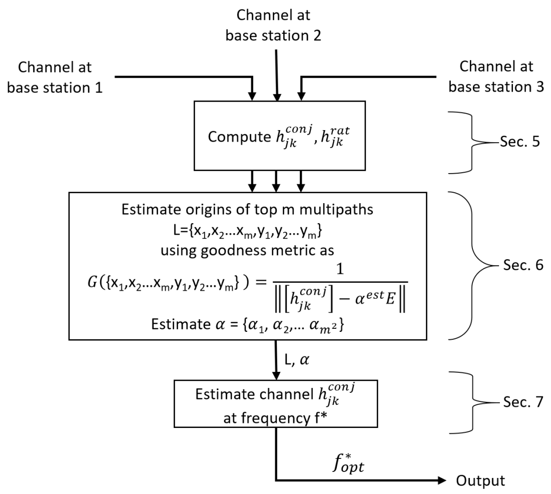
Channels are Sparse and Changing: Key to our optimization above is an accurate estimate of the number of dominant signal paths m. Choosing a small number of signal paths would lead to inefficiency and a poor overall goodness-of-fit relative to the observed wireless channel. However, choosing a large number of signal paths leads to over-fitting, or requires large number of base stations, and eventually, a poor estimate of the optimal frequency to operate on. Fortunately, our results in Sec. 10.2 demonstrate that the number of dominant signal paths in practical outdoor settings is small, a median of 2, beyond which we tend to over-fit. Furthermore, given a number of base stations, estimating a certain number of dominant paths give the best results. We analyze this optimum sparsity in Sec. 10.3 empirically and use the appropriate m for performing the optimization. In Sec. 10.2 we show that even though multipath is sparse, the dominant taps change over the time-scale of minutes causing the optimal frequency to change. Of course, most large reflectors (buildings) do not move to cause this change. Hence, we surmise this is the aggregate effect of one or more smaller static objects (e.g. parked vehicles, objects close to the transmitter/receiver) that move at these time scales.

Multiple, Non-Linear and Fleeting Reflectors: Note while reflecting surfaces may be non-linear in the real world, in our model, we only consider linear reflectors. We thus model the multiple reflections off a non-linear reflector or multiple reflectors as a composite linear reflector. Indeed, while this assumption may sometimes lead to erroneous estimation of reflectors [52] due to increased path length, we see that by expanding the physical size of our search space for virtual sources, the error in estimating multipath is minimal. We also ran simulation based experiments which attempt to estimate multiple reflections with a single reflector over a larger search space (due to larger path distances). The results show that error in finding the virtual source is negligible as long as the peak is 7 dB above noise across the ISM band. Practically, Chime only needs a source which exhibits similar distances to the base stations as the different paths to the base stations. We also ignore fleeting and small reflectors, which occur only for one of the base stations, since they have minimal amortized effect across the received signals of the base stations. Our results in Sec.10.3 show that these assumptions work reasonably well for urban environments.

Finite Reflectors: Our approach above assumes infinite planar reflectors which is not true in real world. To encode the finiteness of the reflectors we can introduce a new parameter β_p which is a boolean vector of length n to each virtual source p where β_{pj}=1 denotes whether the signal from virtual source p reaches base station j. This means that

$$E_{m^2\times n^2} = \left[\frac{\beta_{pj}\beta_{qk}}{d_{pj}d_{qk}}e^{-i2\pi\frac{d_{pj}-d_{qk}}{\lambda}}\right]_{p,q=1,\ldots,m;\ j,k=1,\ldots,n}$$

would be sufficient to model finite reflectors. However, simply looking for all possible β is inefficient. Instead we add two new parameters Φ_k and ψ_k which represent the starting angle and spanning angle of the planar reflector. We can reduce the possible β by imposing practical and geographical constraints. This means we only have to optimize for $2m$ extra parameters instead of nm. We can estimate the β matrix by



Channel at base station 2
Channel at base station 1
Channel at base station 3
Compute $h_{jk}^{conj}, h_{jk}^{rat}$
Sec. 5
Estimate origins of top m multipaths
L={x1,x2...xm,y1,y2...ym}
using goodness metric as
$G(\{x_1,x_2...x_m,y_1,y_2...y_m\}) = \frac{1}{\left\|\left[h_{jk}^{conj}\right] - \alpha^{est}E\right\|}$
Estimate $\alpha = \{\alpha_1, \alpha_2, \ldots \alpha_{m^2}\}$
Sec. 6
L, α
Estimate channel h_{jk}^{conj} at frequency f*
Sec. 7
f^*_{opt}
Output

Figure 7: **Chime**'s algorithm in a nutshell

using the source locations$\{x_k, y_k, z_k\}$ and the angles Φ_k, ψ_k and applying the correct constraints. This means that the number of variables does not increase significantly and can be modeled with additional base stations.

Mobility: While our solution does not consider mobility of client, we believe even with the knowledge of location of the client device, we **cannot** use the reflectors computed during previous run of Chime to assist the next run. This is because LP-WAN devices transmit very rarely (about every 15 minutes or more), which according to our observations in Sec. 3 demonstrate change in multipath of even static clients. This will lead to a complete change in the reflectors of the client device. Thus, under Chime's constraints, we will indeed need to recompute every reflector again.

Extending to Multiple Frequencies To recover the dominant signal paths using the algorithm above, one would need to ensure that $m < n$, i.e. the number of dominant signal paths is sparse and well below the number of base stations in the vicinity of the LP-WAN client. We note that not all these base stations need to be able to decode the client's transmission at the highest rate – they can simply compute wireless channels from the preamble. However, in the instance that too few base stations are available in the vicinity of the client, Chime can improve its performance by measuring wireless channels at more frequency bands, e.g. by requesting the client to hop across a few bands. In effect, the additional measurements across base stations makes sure our optimization in Eqn. 7 is not under-determined.

7 Estimating Optimal Frequency

Having disentangled the multiple signal paths emerging from the client, Chime can estimate an optimal frequency of operation by recombining these signal paths across the various available transmission frequencies. It can then identify the operating frequency by choosing the transmission frequency with the highest signal power, while also accounting for other factors such as noise and interference.

Computing Signal Power The first step to selecting the best operating frequency is to determine the signal power at each frequency band. In particular, given the offset-free channels h^{conj}_{jk} between a pair of base stations (j,k) from Sec. 5 and the multipath propagation characteristics ($\{\alpha_{p,q}\}$ and $\{(x_p, y_p, z_p)\}$) from Sec. 6, we write the offset-free channel $h^{\text{conj}}_{jk,@f}$ at any frequency f and wavelength λ_f as:

$$h^{\text{conj}}_{jk,@f} = [\alpha_{p,q}]_{1\times m^2} E \qquad \text{, where:}$$

$$E = \left[\frac{1}{d_{pj} d_{qk}} e^{-i2\pi \frac{d_{pj}-d_{qk}}{\lambda_f}} \right]_{p,q=1,\dots,m;\ j,k=1,\dots,n}$$

Here, d denotes the distances between virtual sources and base stations as defined in Sec. 6.

Notice that the magnitude of $|h^{\text{conj}}_{jk,@f}|^2$ is simply the product of the signal power from the client at base station i and base station j. However, Chime needs to recover the individual power of the wireless channels at each frequency to compare them across frequencies. Extracting these individual powers from $h^{\text{conj}}_{jk,@f}$ is challenging, because its phase was carefully constructed to remove any hardware impediments.

Chime addresses this challenge by performing its algorithm in Sec. 6 on a second set of input wireless channels. We define these wireless channels, h^{rat}_{jk} as:

$$h^{\text{rat}}_{jk} = \frac{\tilde{h}_{C\to B_k}(t)\tilde{h}_{B_M\to B_j}(t) h_{B_M\to B_k}}{\tilde{h}_{C\to B_j}(t)\tilde{h}_{B_M\to B_k}(t) h_{B_M\to B_j}}$$

Notice that the phase of h^{rat}_{jk} is identical to that of h^{conj}_{jk}, and is therefore also free from phase errors due to hardware impediments of LP-WAN radios. Its magnitude however is **different** – the ratio of the magnitude of the wireless channels to each base stations. One can therefore apply Chime's algorithm (Sec. 6) with h^{rat}_{jk} instead of h^{conj}_{jk} as input and obtain as output the corresponding wireless channel at frequency f: $h^{\text{rat}}_{jk,@f}$. It is then easy to see that the power of the signal from the client to base stations j and k on frequency f is:

$$|h_{C\to B_j,@f}|^2 = h^{\text{conj}}_{jk,@f} / h^{\text{rat}}_{jk,@f} \tag{8}$$

$$|h_{C\to B_k,@f}|^2 = (h^{\text{conj}}_{jk,@f})^* h^{\text{rat}}_{jk,@f} \tag{9}$$

Selecting Optimal Radio Configuration Beyond signal power at the target frequency that Chime computes, data rate is also influenced by ambient noise, interference and attenuation introduced by the transmit/receive chains across frequencies. Fortunately, LP-WAN base stations can easily measure all these quantities as they span a wide band of frequencies [31]. Chime therefore uses these measurements to compute the effective SINR of the client across frequencies to choose the one best optimizing its battery life.

8 Extensions of Chime

While Chime is designed to compute an optimal frequency for an LP-WAN client to conserve battery-life, its approach can be used to complement related problems in LP-WAN:

Coherent Combining: In addition to magnitude, recall that Chime also provides the relative phase of wireless channels between base stations. This is useful in computing the expected wireless channels when the base stations collaborate to coherently combine the received signal across base stations in order to decode them (e.g. Charm [17] performs coherent combining in the LP-WAN context to decode weak transmissions from clients). By knowing both the magnitude and relative phase of wireless channels at each base station across frequencies, Chime can identify the frequency-band for which the expected power of the coherently combined signal will be maximum. Hence, Chime improves the performance of coherent combining in LP-WANs (Sec. 10.5).

Finding Nulls: Just as Chime can find the radio configuration where a client's signal power to any base station is maximum, it can also find frequency where signal power is minimum. This is valuable in *nulling* interference from an unwanted client at a base station by requesting it to transmit at a frequency where interference is lower with one or more base stations. Sec. 10.6 presents results evaluating Chime's performance in finding nulls from a client to base station.

9 Implementation and Evaluation

We implement Chime on Ettus USRP N210s as base stations and reference transmitter for removing phase offsets (see Sec. 5).These base stations measure phase based on our customized code in UHD to measure phase for Chirp Spread Spectrum modulated data at line rate. We use Semtech SX1276 chips as LoRaWAN client transmitters. Each picks a single frequency from the ones supported by the transmitter and transmits a small "chirp" for the base stations to hear. The master base station (USRP N210) is designed to transmit on an adjacent band all the time for convenience of implementation(see Sec. 5.2). We set the client spreading factor to 10 bits per symbol and the bandwidth to 125KHz (standard mode of operation). Each base station has a reliable link to the cloud via a wired backend. Chime's code is implemented in MATLAB/C++ using an in-house UHD-compatible LoRaWAN demodulator and processes the received wireless channels across base stations at the cloud. We only consider infinite length reflectors to evaluate our system. Note that we perform coherent combining across base stations only for Sec. 10.5 where we combine Charm with Chime.

Wide-Area Deployment: Unless specified otherwise, we evaluate Chime over four months across CMU campus and surrounding neighborhoods spanning an area of 0.5km $\times$ 0.7km in Pittsburgh leading to complex multipath scenarios as shown in Fig. 8. Our deployment consists of 11 LP-WAN

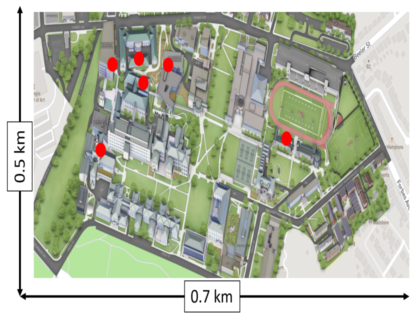


Figure 8: **Chime Deployment**: Red circles denote base station locations

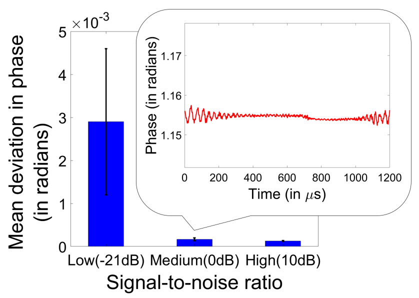


Figure 9: **Phase Stability**: Phase of offset-free channel h_{jk}^{conj} in multipath-rich scenarios is stable across SINRs

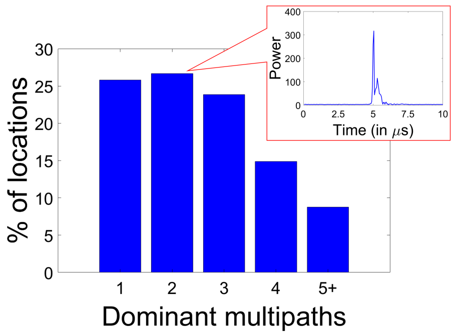


Figure 10: **Multipath Sparsity:** Histogram of # dominant paths shows sparsity of multipath in urban environment

base stations serving different areas, all placed in different buildings – 5 indoors and 6 outdoors. The campus has a variety of tall buildings, trees, other large occlusions and hilly terrain. Our frequencies of operation include the 915 MHz ISM band and some bands in 500 MHz TV white spaces (FCC experimental hardware license). We deploy up to 30 static LoRaWAN clients at various locations (changing every few days) to collect thousands of wireless channel traces across distances relative to the base stations. While each client is not mobile, we do consider a dynamic environment. Each client transmits at a rate of 5-15 packets per hour. Further, clients chose an arbitrary frequency of operation for their initial association packet. Note that our experiments in Sec. 10.2 are in a 0.36 km^2 downtown area of Pittsburgh to study multipath (described further in Sec. 10.2).

Ground Truth & Baseline: We obtain ground truth by making clients hop on all frequencies to find an optimal one. However, only the wireless channel corresponding to a single packet on one frequency band is provided to Chime, unless stated otherwise. We compare Chime against three baseline systems: (1) Standard LoRaWAN which chooses initial frequency arbitrarily; (2) Interpolation across frequency (as described in Sec. 3), when data across multiple frequencies is available; (3) Charm, a system that performs coherent combining across base stations [17].

Runtime: Our current implementation takes $\sim$ 31 sec to explore the search space of reflectors on a desktop with Core-i7 8700K and Nvidia GTX 1060 GPU with 64 GB RAM where E matrices are prefetched in memory for search space of virtual sources. This could be significantly optimized with prior knowledge of the reflectors (topography) or parallelization on a GPU cluster – a task for future work.

10 Experimental Results

10.1 Stability of Phase

Setup: An LP-WAN transmitter is moved across 25 locations in our wide-area testbed and multiple traces are collected from base stations spread across 4 months for static clients. We remove the phase offsets and plot the mean and standard deviation of the instability (standard deviation) in the phase of the offset-free channel (Sec. 5) across pairs of base stations for various SINRs.

Results: Fig. 9 shows the phase measurements of the offset-free channel are stable across pairs of base stations with a mean standard deviation of less than 5×10^{-3} even at SINRs as low as -21 dB. This validates the stability in measurement of the phase of offset-free channels at low SINRs.

10.2 Multipath in Urban Environments

We next study the multipath in the downtown of a large city in the U.S. to validate the sparsity assumption in Sec. 6.1.

Setup: We have a base station transmit wide band chirps of 20 MHz moved over a path length of 5 km in a urban downtown environment. Another base station is used to receive these signals. We then collect data from over 600 different GPS-tagged locations over 0.36 km^2. We correlate with transmitted chirp to estimate the number of taps in the signal. We also keep a transmitter-receiver pair 600 m away in a NLOS suburban environment to evaluate the change in sparsity of multipath and the associated channels over time.[3]

Results: Fig. 10 shows that almost 77% of locations have less than 3 dominant taps in the wireless channel affecting the signal, showing the channel is predominantly sparse as we assume. We also note that at least one of these dominant taps change over time scales of a few minutes, even for static clients. We surmise this is due to some smaller static reflectors in the environment moving gradually over time in aggregate leading to a small number of gradually moving taps. Fig. 11 shows how long the paths between the client and the base station typically are stable. We define *persistence* of a path as the time until which atleast 80% of the energy received remains within the original path components. We see that with 90% likelihood the sparse multipath changes within 10 mins. If we ignore the most dominant path, we see that secondary reflectors change even faster.

[3]Our data and code are available at [1].

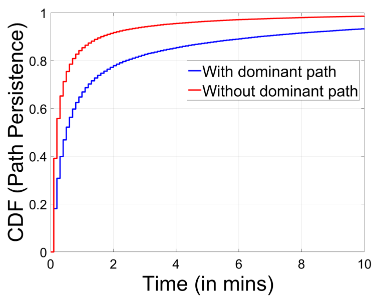


Figure 11: **Path Persistence**: Sparse multipath is unstable across minutes

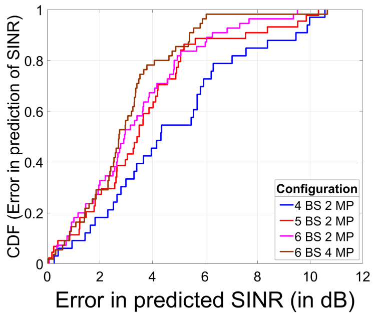


Figure 12: **SINR Goodness-of-Fit**: CDF of predicted vs. actual SINR across base stations

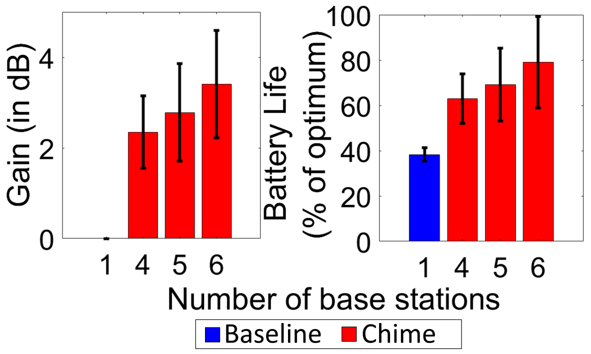


Figure 13: (Left) Gain in SINR(dB) by using Chime vs. median frequency of operation; (Right) Battery life of Chime vs. temporal interpolation technique

10.3 Chime's Gains across Base Stations

We demonstrate the gains achieved by using Chime for identifying an optimal frequency of operation.

Setup: We collect 20 measurements of 100 packets each, spread across 3 months from 5 locations across campus at six base stations at a given frequency f_i. Using these packets, we compute the offset-free channel for each of the base station pairs. We then apply Chime's algorithm to compute an optimal frequency-of-operation. We compute the **gain** (in dB) as the improvement of SINR at the computed operation frequency vs. the median SINR across all possible frequencies. Finally, we measure the improvement in the **battery life** of LP-WAN transmitters due to lower transmission time by using Chime as the percentage of maximum battery life achievable by choosing the optimum frequency of operation. The results are averaged over choice of initial frequency.

Sparsity: As we increase the number of base stations, more and more complex multipath patterns emerge. This is to be expected, given that more base stations are influenced by a larger number of reflectors. This means that 2 multipath sources are not enough to correctly estimate the complex multipath patterns and hence more multipath sources are required to assess the optimum frequency of operation. This can usually be rectified by adding more variables (estimating more sources) which can result in a better fit for the equations. The table shows the median optimum sparsity vs. # base stations:

# Base stations	**Optimal Sparsity**
4	2 sources
5	3 sources
6	4 sources

SINR Prediction: Next, we measure how accurately Chime predicts the accurate SINR of the optimal frequency of operation, across the 915 MHz ISM band. Specifically, we compute the CDF of the difference in SINR between the predicted and actual SINR at the optimal frequency of operation. Fig. 12 plots the results across number of base stations with only 2.7 dB of difference (median) with 6 base stations and 4 multipath(MP) sources considered. To put this in perspective, the SINR at an arbitrary frequency would differ from the optimal by as much as 6.1 dB (median). Our results once again validate Chime's sparsity assumptions and our modeling approach. It shows that the gap between the association packet and transmission ($\sim$10-15 ms) is too short for environmental dynamism to change the channels for static clients.

SINR Gain: Next we analyze the gain in SINR achieved by Chime with increasing number of base stations. Our baseline for the gain is the median frequency-of-operation which emulates choosing an operation frequency at random. As shown in Fig. 13, we achieve a gain of about 2.4 dB with 4 base stations which increases as we increase the number of base stations (with optimum sparsity). With 6 base stations, we achieve a mean increase in the SINR of about 3.4 dB.

Battery Life Gain: Finally, we compare the battery life[4] achieved by Chime with that of choosing frequency of operation based on temporal interpolation. As shown in Fig. 13, we see a stark improvement of 107% in the battery life using Chime which provides a mean of 79% of the optimum over the baseline approaches. This result shows that Chime can provide high gains for dense urban deployments.

These gains in signal power allow transmitters to send at faster rates and reducing the transmission time of the LP-WAN clients. We use methodology explained in Sec. 3 to estimate the expected battery life of the client when streaming sensed data at the optimal data rate to the base station. As shown in Fig. 14, we see a 230% increase in the battery life of the LP-WAN transmitters over the median frequency of operation which is significant for rarely transmitting devices whose lifetime increases from 2.5 years to 8.2 years.

10.4 Chime's Gains across Frequency

We study the gain in SINR and improvement in data rate that can be obtained by sampling more frequencies to further help the base stations to find an optimal frequency using Sec. 6.2.

Setup: We collect phases from 6 receiver base stations at frequencies ranging from 902-928 MHz with an interval of 500 kHz. The frequencies chosen in each case for training are randomized to ensure correctness and the gains obtained in each case are averaged across 5 client locations across

[4]Battery Life estimates derived from prior energy models (see Sec.3)

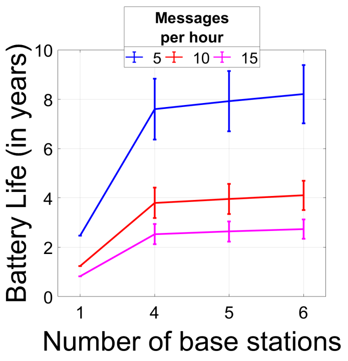


Figure 14: **Gain in battery life across # messages per hour**: Battery life increases 1.4-5.7 years for LP-WAN clients

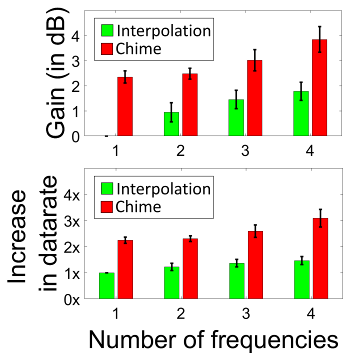


Figure 15: **Gain in SINR and improvement in datarate vs. interpolation** for # of frequencies used for training

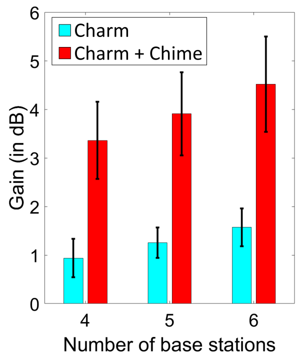


Figure 16: **Chime + Charm**: Improvement in Gain(dB) when Charm is assisted by Chime

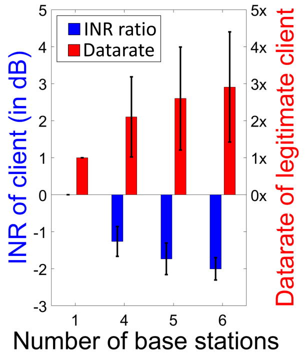


Figure 17: **Nulling of unwanted interference** leads to improved data rate for legitimate client with Chime

multiple weeks. We compute the improvement in data rates achieved due to higher signal strength. As we are sampling multiple frequencies, our baseline will be the spectral interpolation using these frequencies (as described in Sec. 3).

Results: We observe a steady increase in gain with increasing number of frequencies used for training which improves battery-life. The improvement is significantly more than that of the baseline. An important side-benefit of Chime is the improvement in data rate which also progressively increases. As the SINR of the received signal improves, it enables clients to transmit at faster data rates. While LP-WAN clients are infrequent and low-rate transmitters, this improves overall spectrum utilization in congested large-scale deployments.

10.5 Chime with Coherent Combining

In this experiment, we measure Chime's performance in improving Charm's [17] capability of coherent combining.

Setup: We perform the same experiment as Sec. 10.3. However, to compute the frequency-of-operation, we optimize for the sum of the SINR at the base stations instead of an individual base station. Then, we coherently combine the signals at that frequency as shown in [17]. The base line is näive Charm [17] which chooses a frequency randomly.

Results: Fig. 16 shows a median SINR increase of 4.5 dB with six base stations which can significantly improve the battery life of LP-WAN clients in urban environments. As expected, the improvement is much better than that by choosing a random frequency of operation by about 2.5-3 dB.

10.6 Can Chime Null Interference?

This section predicts nulls, i.e. a bad frequency of transmission for an interfering client to a given base station to provide improvement in signal strength of legitimate client.

Setup: We perform the same experiment as Sec. 10.3. We measure the reduction in interference by using Chime to correctly estimate the frequency with the worst channel estimate. We compute the reduction in Interference to Noise Ratio (INR) for a legitimate client in another channel. We measure the resulting gains in data-rate for the legitimate transmitter due to reduction in interference by the interferer.

Results: Fig. 17 shows that we can achieve up to 2.8× gain in the data rates of the legitimate transmitter by allocating the interferer a null frequency. We further show that as we increase the number of base stations, the accuracy of estimating nulls increases which shows that we can get better and better gains for the legitimate transmitter.

11 Conclusion and Future Work

This paper presents Chime, a system that allows an LP-WAN client to choose its optimal frequency simply by sending a single packet on one frequency band. Chime achieves this by analyzing the paths signals traverse from the client to distributed and coordinated base stations. Chime was evaluated in a campus-scale testbed, leading to a median battery life increase of 1.4-5.7 years over commodity LP-WANs.

While Chime's emphasis is on optimal frequency, we believe it provides the building blocks for a comprehensive interference management and distributed MIMO system built for LP-WANs. Designing such an end-to-end system to provide enormous battery savings to low-power clients, while respecting their hardware limitations remains an important problem for future work.

Acknowledgments

This work was supported by Kavčić-Moura award and National Science Foundation grants 1823235, 1718435 and 1837607. We extend warm gratitude to Prof. Srinivasa Narasimhan, Angy Malloy and Robotics Institute at CMU for allowing us to setup remote base stations for wide-scale deployment. Finally, the authors would like to thank our shepherd Lin Zhong, anonymous NSDI reviewers, Vyas Sekar, Peter Steenkiste, Atulya, Adhishree and members of WiSe and WiTech lab for their constructive feedback and support.

References

[1] Chime Code and Link to Data. https://github.com/AkshayGadre/ChimeNSDI2020/, accessed Aug 23, 2019.

[2] LoRaWAN – What is it? A Technical Overview of LoRa and LoRaWAN. https://www.lora-alliance.org/portals/0/documents/whitepapers/LoRaWAN101.pdf, accessed Jan 10, 2019.

[3] LoRaWAN Capacity Trial In Dense Urban Environment. https://www.smart-city-solutions.de/wp-content/uploads/2018/04/machineQ_LoRaWan_Capacity_Trial.pdf, accessed Jan 10, 2019.

[4] Semtech SX1276 datasheet. https://www.semtech.com/uploads/documents/DS_SX1276-7-8-9_W_APP_V5.pdf, accessed Jun 15, 2018.

[5] Long Term Evolution for Machines: LTE-M. https://www.gsma.com/iot/long-term-evolution-machine-type-communication-lte-mtc-cat-m1/, accessed Mar 3, 2018.

[6] Ferran Adelantado, Xavier Vilajosana, Pere Tuset-Peiro, Borja Martinez, Joan Melia-Segui, and Thomas Watteyne. Understanding the limits of LoRaWAN. *IEEE Communications magazine*, 55(9):34–40, 2017.

[7] Emekcan Aras, Nicolas Small, Gowri Sankar Ramachandran, Stéphane Delbruel, Wouter Joosen, and Danny Hughes. Selective jamming of LoRaWAN using commodity hardware. In *EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 363–372, 2017.

[8] Waheed U Bajwa, Jarvis Haupt, Akbar M Sayeed, and Robert Nowak. Compressed channel sensing: A new approach to estimating sparse multipath channels. *Proceedings of the IEEE*, 98(6):1058–1076, 2010.

[9] A Canavitsas, LAR Silva Mello, and M Grivet. White space prediction technique for cognitive radio applications. In *IEEE Microwave & Optoelectronics Conference (IMOC)*, pages 1–5, 2013.

[10] Marco Centenaro, Lorenzo Vangelista, Andrea Zanella, and Michele Zorzi. Long-range communications in unlicensed bands: The rising stars in the IoT and smart city scenarios. *IEEE Wireless Communications*, 23(5):60–67, 2016.

[11] Gerard G Cervello, Sunghyun Choi, Stefan Mangold, and Amjad Ali Soomro. Dynamic channel selection scheme for IEEE 802.11 WLANs, January 10 2006. US Patent 6,985,465.

[12] H. Chen, L. Liu, T. Novlan, J. D. Matyjas, B. L. Ng, and J. Zhang. Spatial spectrum sensing-based device-to-device cellular networks. *IEEE Transactions on Wireless Communications*, 15(11):7299–7313, Nov 2016.

[13] Min Chen, Yiming Miao, Yixue Hao, and Kai Hwang. Narrow band internet of things. *IEEE Access*, 5:20557–20577, 2017.

[14] W. Cheng, X. Zhang, and H. Zhang. Full-Duplex Spectrum-Sensing and MAC-Protocol for Multichannel Nontime-Slotted Cognitive Radio Networks. *IEEE Journal on Selected Areas in Communications*, 33(5):820–831, May 2015.

[15] Junil Choi, David J Love, and Patrick Bidigare. Downlink training techniques for FDD massive MIMO systems: Open-loop and closed-loop training with memory. *IEEE Journal of Selected Topics in Signal Processing*, 8(5):802–814, 2014.

[16] Federal Communications Commission. FCC Adopts Rules for Unlicensed Use of Television White Spaces. https://www.fcc.gov/document/fcc-adopts-rules-unlicensed-use-television-white-spaces, 2008.

[17] Adwait Dongare, Revathy Narayanan, Akshay Gadre, Anh Luong, Artur Balanuta, Swarun Kumar, Bob Iannucci, and Anthony Rowe. Charm: Exploiting geographical diversity through coherent combining in Low-power Wide-area Networks. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 60–71, 2018.

[18] Zheng Du, Xuegui Song, Julian Cheng, and Norman C Beaulieu. Maximum likelihood based channel estimation for macrocellular OFDM uplinks in dispersive time-varying channels. *IEEE Transactions on Wireless Communications*, 10(1):176–187, 2011.

[19] Rashad Eletreby, Diana Zhang, Swarun Kumar, and Osman Yağan. Empowering Low-Power Wide Area Networks in Urban Settings. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 309–321, 2017.

[20] FCC. Second Report and Order and Memorandum Opinion and Order. *Tech. Rep.*, 2010.

[21] Jose A Gutierrez, Marco Naeve, Ed Callaway, Monique Bourgeois, Vinay Mitter, and Bob Heile. IEEE 802.15 4: a developing standard for low-power low-cost wireless personal area networks. *IEEE network*, 15(5):12–19, 2001.

[22] Ezzeldin Hamed, Hariharan Rahul, Mohammed A Abdelghany, and Dina Katabi. Real-time distributed MIMO systems. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 412–425, 2016.

[23] Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota. Netscatter: Enabling large-scale backscatter networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[24] Hyoil Kim and Kang G Shin. Efficient discovery of spectrum opportunities with MAC-layer sensing in cognitive radio networks. *IEEE transactions on mobile computing*, 7(5):533–545, 2008.

[25] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[26] Manikanta Kotaru, Kiran Joshi, Dinesh Bharadia, and Sachin Katti. Spotfi: Decimeter level localization using wifi. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 269–282, 2015.

[27] Jansen C Liando, Amalinda Gamage, Agustinus W Tengourtius, and Mo Li. Known and Unknown Facts of LoRa: Experiences from a Large-scale Measurement Study. *ACM Transactions on Sensor Networks (TOSN)*, 15(2):16, 2019.

[28] Rúben Oliveira, Lucas Guardalben, and Susana Sargento. Long range communications in urban and rural environments. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 810–817, 2017.

[29] Chunyi Peng, Haitao Zheng, and Ben Y. Zhao. Utilization and fairness in spectrum assignment for opportunistic spectrum access. *Mobile Networks Applications*, 11(4):555–576, August 2006.

[30] Yao Peng, Longfei Shangguan, Yue Hu, Yujie Qian, Xianshang Lin, Xiaojiang Chen, Dingyi Fang, and Kyle Jamieson. PLoRa: Passive Long-Range Data Networks from Ambient LoRa Transmissions. *ACM Special Interest Group on Data Communication(SIGCOMM)*, 2018.

[31] Tara Petrić, Mathieu Goessens, Loutfi Nuaymi, Laurent Toutain, and Alexander Pelov. Measurements, performance and analysis of LoRa FABIAN, a real-world implementation of LPWAN. In *IEEE International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–7, 2016.

[32] R. Grech. Semtech and Comcast's machineQ Announce LoRaWAN Network Availability in 10 Cities. https://www.semtech.com/company/press/semtech-and-comcasts-machineq-announce-lorawan-network-availability 2018.

[33] Hariharan Rahul, Swarun Kumar, and Dina Katabi. MegaMIMO: Scaling Wireless Capacity with User Demands. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, August 2012.

[34] Xiongbin Rao and Vincent KN Lau. Distributed compressive CSIT estimation and feedback for FDD multi-user massive MIMO systems. *IEEE Transactions on Signal Processing*, 62(12):3261–3271, 2014.

[35] Rapeepat Ratasuk, Benny Vejlgaard, Nitin Mangalvedhe, and Amitava Ghosh. NB-IoT system for M2M communication. In *IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–5, 2016.

[36] S. Marek. Comcast Will Test LoRaWAN IoT Networks in Two Markets. https://www.sdxcentral.com/articles/news/comcast-will-test-lora-iot-network-two-markets/2016/10/, 2016.

[37] Jaroslaw Sadowski. Measurement of coherence bandwidth in uhf radio channels for narrowband networks. *International Journal of Antennas and Propagation*, 2015, 2015.

[38] Shweta Sagari, Samuel Baysting, Dola Saha, Ivan Seskar, Wade Trappe, and Dipankar Raychaudhuri. Coordinated dynamic spectrum management of LTE-U and Wi-Fi networks. In *IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, pages 209–220, 2015.

[39] Ramon Sanchez-Iborra and Maria-Dolores Cano. State of the Art in LP-WAN Solutions for Industrial IoT Services. *Sensors*, 16(5):708, 2016.

[40] Mamoru Sawahashi, Yoshihisa Kishiyama, Akihito Morimoto, Daisuke Nishikawa, and Motohiro Tanno. Coordinated multipoint transmission/reception techniques for LTE-advanced [Coordinated and Distributed MIMO]. *IEEE Wireless Communications*, 17(3), 2010.

[41] Souvik Sen, Bozidar Radunovic, Jeongkeun Lee, and Kyu-Han Kim. CSpy: finding the best quality channel without probing. In *ACM International Conference on Mobile Computing & Networking (MobiCom)*, pages 267–278, 2013.

[42] C Shepard, H Yu, N Anand, L Li, T Marzetta, YR Yang, and L Zhong. Argos: Practical base stations with large-scale multi-user beamforming. *ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 53–64, 2012.

[43] Min Soo Sim, Jeonghun Park, Chan-Byoung Chae, and Robert W Heath. Compressed channel feedback for correlated massive MIMO systems. *Journal of Communications and Networks*, 18(1):95–104, 2016.

[44] N. Sornin, M. Luis, T. Eirich, T. Kramp, and O. Hersent. LoRaWAN Specification. pages 1–82, 2015.

[45] Vamsi Talla, Mehrdad Hessar, Bryce Kellogg, Ali Najafi, Joshua R Smith, and Shyamnath Gollakota. LoRa backscatter: Enabling the vision of ubiquitous connectivity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 1(3):105, 2017.

[46] Lei Tang, Yanjun Sun, Omer Gurewitz, and David B. Johnson. EM-MAC: A Dynamic Multichannel Energy-efficient MAC Protocol for Wireless Sensor Networks. In *ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*. ACM, 2011.

[47] David Tse and Pramod Viswanath. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[48] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta N Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. FarmBeats: An IoT Platform for Data-Driven Agriculture. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 515–529, 2017.

[49] Deepak Vasisht, Swarun Kumar, and Dina Katabi. Decimeter-level localization with a single wifi access point. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 16, pages 165–178, 2016.

[50] Deepak Vasisht, Swarun Kumar, Hariharan Rahul, and Dina Katabi. Eliminating channel feedback in next-generation cellular networks. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 398–411, 2016.

[51] Y-P Eric Wang, Xingqin Lin, Ansuman Adhikary, Asbjorn Grovlen, Yutao Sui, Yufei Blankenship, Johan Bergman, and Hazhir S Razaghi. A primer on 3GPP narrowband Internet of Things. *IEEE Communications Magazine*, 55(3):117–123, 2017.

[52] Teng Wei, Anfu Zhou, and Xinyu Zhang. Facilitating robust 60 GHz network deployment by sensing ambient reflectors. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 213–226, 2017.

[53] Yaxiong Xie, Zhenjiang Li, and Mo Li. Precise Power Delay Profiling with Commodity WiFi. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 53–64, 2015.

[54] Jie Xiong and Kyle Jamieson. Arraytrack: A fine-grained indoor location system. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 71–84, 2013.

[55] Yi Xu, Guosen Yue, and Shiwen Mao. User grouping for massive MIMO in FDD systems: New design methods and analysis. *IEEE Access*, 2:947–959, 2014.

[56] Tevfik Yucek and Huseyin Arslan. A survey of spectrum sensing algorithms for cognitive radio applications. *IEEE communications surveys & tutorials*, 11(1):116–130, 2009.

[57] Y. Zeng and Y. C. Liang. Eigenvalue-based spectrum sensing algorithms for cognitive radio. *IEEE Transactions on Communications*, 57(6):1784–1793, June 2009.

[58] Per Zetterberg. Experimental investigation of tdd reciprocity-based zero-forcing transmit precoding. *EURASIP Journal on Advances in Signal Processing*, 2011(1):137541, 2011.

[59] D. Zhang, Z. Chen, J. Ren, N. Zhang, M. K. Awad, H. Zhou, and X. S. Shen. Energy-Harvesting-Aided Spectrum Sensing and Data Transmission in Heterogeneous Cognitive Radio Sensor Network. *IEEE Transactions on Vehicular Technology*, 66(1):831–843, Jan 2017.

[60] Tan Zhang, Ning Leng, and Suman Banerjee. A vehicle-based measurement framework for enhancing whitespace spectrum databases. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 17–28, 2014.

[61] JC Zuniga and B Ponsard. Sigfox System Description. *LPWAN IETF97, Nov. 14th*, 2016.

ABC: A Simple Explicit Congestion Controller for Wireless Networks

Prateesh Goyal[1], Anup Agarwal[2]*, Ravi Netravali[3]†, Mohammad Alizadeh[1], Hari Balakrishnan[1]
[1]MIT CSAIL, [2]CMU, [3]UCLA

Abstract

We propose *Accel-Brake Control* (ABC), a simple and deployable explicit congestion control protocol for network paths with time-varying wireless links. ABC routers mark each packet with an "accelerate" or "brake", which causes senders to slightly increase or decrease their congestion windows. Routers use this feedback to quickly guide senders towards a desired target rate. ABC requires no changes to header formats or user devices, but achieves better performance than XCP. ABC is also incrementally deployable; it operates correctly when the bottleneck is a non-ABC router, and can coexist with non-ABC traffic sharing the same bottleneck link. We evaluate ABC using a Wi-Fi implementation and trace-driven emulation of cellular links. ABC achieves 30-40% higher throughput than Cubic+Codel for similar delays, and 2.2× lower delays than BBR on a Wi-Fi path. On cellular network paths, ABC achieves 50% higher throughput than Cubic+Codel.

1 Introduction

This paper proposes a new explicit congestion control protocol for network paths with wireless links. Congestion control on such paths is challenging because of the rapid time variations of the link capacity. Explicit control protocols like XCP [29] and RCP [43] can in theory provide superior performance on such paths compared to end-to-end [10, 13, 14, 16, 23, 24, 46, 50] or active queue management (AQM) [36, 37] approaches (§2). Unlike these approaches, explicit control protocols enable the wireless router to directly specify a target rate for the sender, signaling both rate decreases and rate increases based on the real-time link capacity.

However, current explicit control protocols have two limitations, one conceptual and the other practical. First, existing explicit protocols were designed for fixed-capacity links; we find that their control algorithms are sub-optimal on time-varying wireless links. Second, they require major changes to packet headers, routers, and endpoints to deploy on the Internet.

Our contribution is a simple and deployable protocol, called Accel-Brake Control (ABC), that overcomes these limitations, building on concepts from a prior position paper [22]. In ABC (§3), a wireless router marks each packet with one bit of feedback corresponding to either *accelerate* or *brake* based on a measured estimate of the current link rate. Upon receiving this feedback via an ACK from the receiver, the sender increases its window by one on an accelerate (sends two packets in response to the ACK), and decreases it by one on a brake (does not send any packet). This simple mechanism allows the router to signal a large dynamic range of window size changes within one RTT: from throttling the window to 0, to doubling the window.

Central to ABC's performance is a novel control algorithm that helps routers provide very accurate feedback on time-varying links. Existing explicit schemes like XCP and RCP calculate their feedback by comparing the current *enqueue rate* of packets to the link capacity. An ABC router, however, compares the *dequeue rate* of packets from its queue to the link capacity to mark accelerates or brakes. This change is rooted in the observation that, for an ACK-clocked protocol like ABC, the current dequeue rate of packets at the router provides an accurate prediction of the future incoming rate of packets, one RTT in advance. In particular, if the senders maintain the same window sizes in the next RTT, they will send one packet for each ACK, and the incoming rate in one RTT will be equal to the current dequeue rate. Therefore, rather than looking at the current enqueue rate, the router should signal changes based on the anticipated enqueue rate in one RTT to better match the link capacity. The impact of this subtle change is particularly significant on wireless links, since the enqueue and dequeue rates can differ significantly when the link capacity varies.

ABC also overcomes the deployability challenges of prior explicit schemes, since it can be implemented on top of the existing explicit congestion notification (ECN) [41] infrastructure. We present techniques that enable ABC to co-exist with non-ABC routers, and to share bandwidth fairly with legacy flows traversing a bottleneck ABC router (§4).

We have implemented ABC on a commodity Wi-Fi router running OpenWrt [18]. Our implementation (§5.1) reveals an important challenge for implementing explicit protocols on wireless links: how to determine the link rate for a user at a given time? The task is complicated by the intricacies of the Wi-Fi MAC's batch scheduling and block acknowledgements. We develop a method to estimate the Wi-Fi link rate and demonstrate its accuracy experimentally. For cellular links, the 3GPP standard [1] shows how to estimate the link rate; our evaluation uses emulation with cellular packet traces.

We have experimented with ABC in several wireless network settings. Our results are:

1. In Wi-Fi, compared to Cubic+Codel, Vegas, and Copa, ABC achieves 30-40% higher throughput with similar delays. Cubic, PCC Vivace-latency and BBR incur 70%–6× higher 95th percentile packet delay with similar throughput.
2. The results in emulation over 8 cellular traces are

*Work done largely while a visiting student at MIT CSAIL.
†Work done largely while a PhD student at MIT CSAIL.

Scheme	Norm. Utilization	Norm. Delay (95%)
ABC	1 (78%)	1 (242ms)
XCP	0.97	2.04
Cubic+Codel	0.67	0.84
Copa	0.66	0.85
Cubic	1.18	4.78
PCC-Vivace	1.12	4.93
BBR	0.96	2.83
Sprout	0.55	1.08
Verus	0.72	2.01

summarized below. Despite relying on single-bit feedback, ABC achieves 2× lower 95th percentile packet delay compared to XCP.

3. ABC bottlenecks can coexist with both ABC and non-ABC bottlenecks. ABC flows achieve high utilization and low queuing delays if the bottleneck router is ABC, while switching to Cubic when the bottleneck is a non-ABC router.
4. ABC competes fairly with both ABC and non-ABC flows. In scenarios with both ABC and non-ABC flows, the difference in average throughput of ABC and non-ABC flows is under 5%.

2 Motivation

Link rates in wireless networks can vary rapidly with time; for example, within one second, a wireless link's rate can both double and halve [46].[1] These variations make it difficult for transport protocols to achieve both high throughput and low delay. Here, we motivate the need for explicit congestion control protocols that provide feedback to senders on both rate increases and decreases based on direct knowledge of the wireless link rate. We discuss why these protocols can track wireless link rates more accurately than end-to-end and AQM-based schemes. Finally, we discuss deployment challenges for explicit control protocols, and our design goals for a deployable explicit protocol for wireless links.

Limitations of end-to-end congestion control: Traditional end-to-end congestion control schemes like Cubic [23] and NewReno [24] rely on packet drops to infer congestion and adjust their rates. Such schemes tend to fill up the buffer, causing large queuing delays, especially in cellular networks that use deep buffers to avoid packet loss [46]. Fig. 1a shows performance of Cubic on an LTE link, emulated using a LTE trace with Mahimahi [35]. The network round-trip time is 100 ms and the buffer size is set to 250 packets. Cubic causes significant queuing delay, particularly when the link capacity drops.

Recent proposals such as BBR [14], PCC-Vivace [16] and Copa [10] use RTT and send/receive rate measurements to estimate the available link rate more accurately. Although these schemes are an improvement over loss-based schemes, their performance is far from optimal on highly-variable links. Our experiments show that they either cause excessive queuing or underutilize the link capacity (e.g., see Fig. 7). Sprout [46] and Verus [50] are two other recent end-to-end protocols designed specifically for cellular networks. They also have difficulty tracking the link rate accurately; depending on parameter settings, they can be too aggressive (causing large queues) or too conservative (hurting utilization). For example, Fig. 1b shows how Verus performs on the same LTE trace as above.

The fundamental challenge for any end-to-end scheme is that to estimate the link capacity, it must utilize the link fully and build up a queue. When the queue is empty, signals such as the RTT and send/receive rate do not provide information about the available capacity. Therefore, in such periods, all end-to-end schemes must resort to some form of "blind" rate increase. But for networks with a large dynamic range of rates, it is very difficult to tune this rate increase correctly: if it is slow, throughput suffers, but making it too fast causes overshoots and large queuing delays.[2] For schemes that attempt to limit queue buildup, periods in which queues go empty (and a blind rate increase is necessary) are common; they occur, for example, following a sharp increase in link capacity.

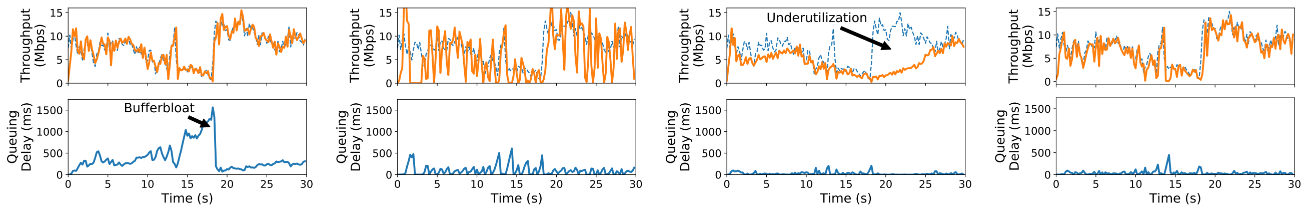
AQM schemes do not signal increases: AQM schemes like RED [19], PIE [37] and CoDel [2] can be used to signal congestion (via ECN or drops) before the buffer fills up at the bottleneck link, reducing delays. However, AQM schemes do not signal rate increases. When capacity increases, the sender must again resort to a blind rate increase. Fig. 1c shows how CoDel performs when the sender is using Cubic. Cubic+CoDel reduces delays by 1 to 2 orders of magnitude compared to Cubic alone but leaves the link underutilized when capacity increases.

Thus, we conclude that, both end-to-end and AQM-based schemes will find it difficult to track time-varying wireless link rates accurately. Explicit control schemes, such as XCP [29] and RCP [43] provide a compelling alternative. The router provides multiple bits of feedback per packet to senders based on direct knowledge of the wireless link capacity. By telling senders precisely how to increase or decrease their rates, explicit schemes can quickly adapt to time-varying links, in principle, within an RTT of link capacity changes.

Deployment challenges for explicit congestion control: Schemes like XCP and RCP require major changes to packet headers, routers, and endpoints. Although the changes are technically feasible, in practice, they create significant deployment challenges. For instance, these protocols require new packet fields to carry multi-bit feedback information. IP or TCP options could in principle be used for these fields. But many wide-area routers drop packets with IP options [20], and using TCP options creates problems due to middleboxes [25] and IPSec encryption [30]. Another important challenge is co-existence with legacy routers and legacy transport

[1]We define the link rate for a user as the rate that user can achieve if it keeps the bottleneck router backlogged (see §5).

[2]BBR attempts to mitigate this problem by periodically increasing its rate in short pulses, but our experiments show that BBR frequently overshoots the link capacity with variable-bandwidth links, causing excessive queuing (see Appendix A).



(a) Cubic (b) Verus (c) Cubic+CoDel (d) ABC

Figure 1: **Performance on a emulated cellular trace —** The dashed blue in the top graph represents link capacity, the solid orange line represents the achieved throughput. Cubic has high utilization but has very high delays (up to 1500 milliseconds). Verus has large rate variations and incurs high delays. Cubic+CoDel reduces queuing delays significantly, but leaves the link underutilized when capacity increases. ABC achieves close to 100% utilization while maintaining low queuing delays (similar to that of Cubic+CoDel).

protocols. To be deployable, an explicit protocol must handle scenarios where the bottleneck is at a legacy router, or when it shares the link with standard end-to-end protocols like Cubic.

Design goals: In designing ABC, we targeted the following properties:

1. *Control algorithm for fast-varying wireless links:* Prior explicit control algorithms like XCP and RCP were designed for fixed-capacity links. We design ABC's control algorithm specifically to handle the rapid bandwidth variations and packet transmission behavior of wireless links (e.g., frame batching at the MAC layer).
2. *No modifications to packet headers:* ABC repurposes the existing ECN [41] bits to signal both increases and decreases to the sender's congestion window. By spreading feedback over a sequence of 1-bit signals per packet, ABC routers precisely control sender congestion windows over a large dynamic range.
3. *Coexistence with legacy bottleneck routers:* ABC is robust to scenarios where the bottleneck link is not the wireless link but a non-ABC link elsewhere on the path. Whenever a non-ABC router becomes the bottleneck, ABC senders ignore window increase feedback from the wireless link, and ensure that they send no faster than their fair share of the bottleneck link.
4. *Coexistence with legacy transport protocols:* ABC routers ensure that ABC and non-ABC flows share a wireless bottleneck link fairly. To this end, ABC routers separate ABC and non-ABC flows into two queues, and use a simple algorithm to schedule packets from these queues. ABC makes no assumptions about the congestion control algorithm of non-ABC flows, is robust to the presence of short or application-limited flows, and requires a small amount of state at the router.

Fig. 1d shows ABC on the same emulated LTE link. Using only one bit of feedback per packet, the ABC flow is able to track the variations in bottleneck link closely, achieving both high throughput and low queuing delay.

3 Design

ABC is a window-based protocol: the sender limits the number of packets in flight to the current congestion window. Window-based protocols react faster to the sudden onset of congestion than rate-based schemes [11]. On a wireless link, when the capacity drops and the sender stops receiving ACKs, ABC will stop sending packets immediately, avoiding further queue buildup. In contrast, a rate-based protocol would take time to reduce its rate and may queue up a large number of packets at the bottleneck link in the meantime.

ABC senders adjust their window size based on explicit feedback from ABC routers. An ABC router uses its current estimate of the link rate and the queuing delay to compute a *target rate*. The router then sets one bit of feedback in each packet to guide the senders towards the target rate. Each bit is echoed to a sender by a receiver in an ACK, and it signals either a one-packet increase ("accelerate") or a one-packet decrease ("brake") to the sender's congestion window.

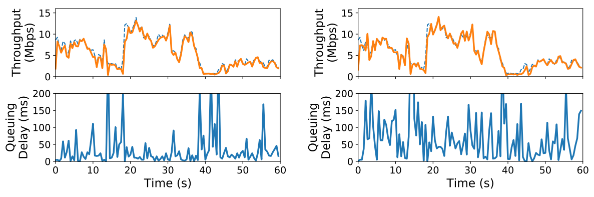
3.1 The ABC Protocol

We now present ABC's design starting with the case where all routers are ABC-capable and all flows use ABC. We later discuss how to extend the design to handle non-ABC routers and scenarios with competing non-ABC flows.

3.1.1 ABC Sender

On receiving an "accelerate" ACK, an ABC sender increases its congestion window by 1 packet. This increase results in two packets being sent, one in response to the ACK and one due to the window increase. On receiving a "brake," the sender reduces its congestion window by 1 packet, preventing the sender from transmitting a new packet in response to the received ACK. As we discuss in §4.3, the sender also performs an additive increase of 1 packet per RTT to achieve fairness. For ease of exposition, let us ignore this additive increase for now.

Though each bit of feedback translates to only a small change in the congestion window, when aggregated over an RTT, the feedback can express a large dynamic range of window size adjustments. For example, suppose a sender's window size is w, and the router marks accelerates on a fraction f of packets in that window. Over the next RTT, the sender will receive $w \cdot f$ accelerates and $w - w \cdot f$ brakes. Then, the sender's window size one RTT later will be $w + wf - (w - wf) = 2wf$ packets. Thus, in one RTT, an ABC router can vary the sender's window size between zero ($f = 0$) and double its current value ($f = 1$). The set of achievable



(a) Dequeue (b) Enqueue

Figure 2: **Feedback —** Calculating $f(t)$ based on enqueue rate increases 95th percentile queuing delay by 2×.

window changes for the next RTT depends on the number of packets in the current window w; the larger w, the higher the granularity of control.

In practice, ABC senders increase or decrease their congestion window by the number of newly acknowledged bytes covered by each ACK. Byte-based congestion window modification is a standard technique in many TCP implementations [8], and it makes ABC robust to variable packet sizes and delayed, lost, and partial ACKs. For simplicity, we describe the design with packet-based window modifications in this paper.

3.1.2 ABC Router

Calculating the target rate: ABC routers compute the target rate $tr(t)$ using the following rule:

$$tr(t) = \eta\mu(t) - \frac{\mu(t)}{\delta}(x(t) - d_t)^+, \tag{1}$$

where $\mu(t)$ is the link capacity, $x(t)$ is the observed queuing delay, d_t is a pre-configured delay threshold, η is a constant less than 1, δ is a positive constant (in units of time), and y^+ is $\max(y,0)$. This rule has the following interpretation. When queuing delay is low ($x(t) < d_t$), ABC sets the target rate to $\eta\mu(t)$, for a value of η slightly less than 1 (e.g., $\eta = 0.95$). By setting the target rate a little lower than the link capacity, ABC aims to trade a small amount of bandwidth for large reductions in delay, similar to prior work [7,27,32]. However, queues can still develop due to rapidly changing link capacity and the 1 RTT of delay it takes for senders to achieve the target rate. ABC uses the second term in Equation (1) to drain queues. Whenever $x(t) > d_t$, this term reduces the target rate by an amount that causes the queuing delay to decrease to d_t in at most δ seconds.

The threshold d_t ensures that the target rate does not react to small increases in queuing delay. This is important because wireless links often schedule packets in batches. Queuing delay caused by batch packet scheduling does not imply congestion, even though it occurs persistently. To prevent target rate reductions due to this delay, d_t must be configured to be greater than the average inter-scheduling time at the router.

ABC's target rate calculation requires an estimate of the underlying link capacity, $\mu(t)$. In §5, we discuss how to estimate the link capacity in cellular and WiFi networks, and we present an implementation for WiFi.

Packet marking: To achieve a target rate, $tr(t)$, the router computes the fraction of packets, $f(t)$, that should be marked as accelerate. Assume that the current *dequeue rate* — the rate at which the router transmits packets — is $cr(t)$. If the accelerate fraction is $f(t)$, for each packet that is ACKed, the sender transmits $2f(t)$ packets on average. Therefore, after 1 RTT, the *enqueue rate* — the rate at which packets arrive to the router — will be $2cr(t)f(t)$. To achieve the target rate, $f(t)$ must be chosen such that $2cr(t)f(t)$ is equal to $tr(t)$. Thus, $f(t)$ is given by:

$$f(t) = \min\left\{\frac{1}{2}\cdot\frac{tr(t)}{cr(t)}, 1\right\}. \tag{2}$$

An important consequence of the above calculation is that $f(t)$ is computed based on the *dequeue* rate. Most explicit protocols compare the enqueue rate to the link capacity to determine the feedback (e.g., see XCP [29]).

ABC uses the dequeue rate instead to exploit the ACK-clocking property of its window-based protocol. Specifically, Equation (2) accounts for the fact that when the link capacity changes (and hence the dequeue rate changes), the rate at the senders changes automatically within 1 RTT because of ACK clocking. Fig. 2 demonstrates that computing $f(t)$ based on the dequeue rate at the router enables ABC to track the link capacity much more accurately than using the enqueue rate.

ABC recomputes $f(t)$ on every dequeued packet, using measurements of $cr(t)$ and $\mu(t)$ over a sliding time window of length T. Updating the feedback on every packet allows ABC to react to link capacity changes more quickly than schemes that use periodic feedback updates (e.g., XCP and RCP).

Packet marking can be done deterministically or probabilistically. To limit burstiness, ABC uses the deterministic method in Algorithm 1. The variable `token` implements a token bucket that is incremented by $f(t)$ on each outgoing packet (up to a maximum value `tokenLimit`), and decremented when a packet is marked accelerate. To mark a packet accelerate, `token` must exceed 1. This simple method ensures that no more than a fraction $f(t)$ of the packets are marked accelerate.

```
token = 0;
for each outgoing packet do
    calculate $f(t)$ using Equation (2);
    token = min(token + $f(t)$, tokenLimit);
    if packet marked with accelerate then
        if token > 1 then
            token = token − 1;
            mark accelerate;
        else
            mark brake;
```

Algorithm 1: Packet marking at an ABC router.

Multiple bottlenecks: An ABC flow may encounter multiple ABC routers on its path. An example of such a scenario is when two smartphone users communicate over an ABC-compliant cellular network. Traffic sent from one user to the other will traverse a cellular uplink and cellular downlink, both of which could be the bottleneck. To support such situations, an ABC sender should send traffic at the smallest of the router-computed target rates along their path. To achieve this goal, each packet is initially marked accelerate by the sender. ABC routers may change a packet marked accelerate to a brake, but not vice versa (see Algorithm 1). This rule guarantees that an ABC router can unilaterally reduce the fraction of packets marked accelerate to ensure that its target rate is not exceeded, but it cannot increase this fraction. Hence the fraction of packets marked accelerate will equal the minimum $f(t)$ along the path.

3.1.3 Fairness

Multiple ABC flows sharing the same bottleneck link should be able to compete fairly with one another. However, the basic window update rule described in §3.1.1 is a multiplicative-increase/multiplicative-decrease (MIMD) strategy,[3] which does not provide fairness among contending flows (see Fig. 3a for an illustration). To achieve fairness, we add an additive-increase (AI) component to the basic window update rule. Specifically, ABC senders adjust their congestion window on each ACK as follows:

$$w \leftarrow \begin{cases} w+1+1/w & \text{if accelerate} \\ w-1+1/w & \text{if brake} \end{cases} \tag{3}$$

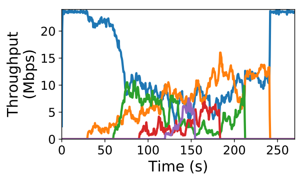
This rule increases the congestion window by 1 packet each RTT, in addition to reacting to received accelerate and brake ACKs. This additive increase, coupled with ABC's MIMD response, makes ABC a multiplicative-and-additive-increase/multiplicative-decrease (MAIMD) scheme. Chiu and Jain [15] proved that MAIMD schemes converge to fairness (see also [5]). Fig. 3b shows how with an AI component, competing ABC flows achieve fairness.

To give intuition, we provide a simple informal argument for why including additive increase gives ABC fairness. Consider N ABC flows sharing a link, and suppose that in steady state, the router marks a fraction f of the packets accelerate, and the window size of flow i is w_i. To be in steady state, each flow must send 1 packet on average for each ACK that it receives. Now consider flow i. It will send $2f+1/w_i$ packets on average for each ACK: $2f$ for the two packets it sends on an accelerate (with probability f), and $1/w_i$ for the extra packet it sends every w_i ACKs. Therefore, to be in steady state, we must have: $2f+1/w_i=1 \implies w_i=1/(1-2f)$. This shows that the steady-state window size for all flows must be the same, since they all observe the same fraction f of accelerates. Hence, with equal RTTs, the flows will have the same throughput, and otherwise their throughput will be inversely proportional to their RTT. Note that the RTT unfairness in ABC is similar to that of schemes like Cubic, for which the throughput of a flow is inversely proportional to its RTT. In §7.5, we show experiments where flows have different RTTs.

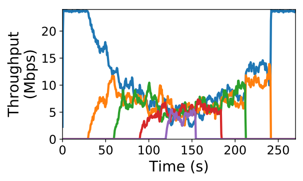
3.1.4 Stability Analysis

ABC's stability depends on the values of η and δ. η determines the target link utilization, while δ controls how long

[3] All the competing ABC senders will observe the same accelerate fraction, f, on average. Therefore, each flow will update its congestion window, w, in a multiplicative manner, to $2fw$, in the next RTT.



(a) ABC w/o AI



(b) ABC with AI

Figure 3: **Fairness among competing ABC flows —** 5 flows with the same RTT start and depart one-by-one on a 24 Mbit/s link. The additive-increase (AI) component leads to fairness.

it will take for a queue to drain. In Appendix C, we prove the following result for a fluid model of the ABC control loop.

Theorem 1. *Consider a single ABC link, traversed by N ABC flows. Let τ be the maximum round-trip propagation delay of the flows. ABC is* globally asymptotically stable *if*

$$\delta > \frac{2}{3} \cdot \tau. \tag{4}$$

Specifically, if $\mu(t)=\mu$ for $t>t_0$ (i.e., the link capacity stops changing after some time t_0), the enqueue/dequeue rate and the queuing delay at the ABC router will converge to certain values r^* and x^* that depend on the system parameters and the number of flows. In all cases: $\eta\mu < r^* \le \mu$.

This stability criterion is simple and intuitive. It states that δ should not be much smaller than the RTT (i.e, the feedback delay). If δ is very small, ABC reacts too forcefully to queue build up, causing under-utilization and oscillations.[4] Increasing δ well beyond $2/3\tau$ improves the stability margins of the feedback loop, but hurts responsiveness. In our experiments, we used $\delta=133$ ms for a propagation RTT of 100 ms.

4 Coexistence

An ABC flow should be robust to presence of non-ABC bottlenecks on its path and share resources fairly with non-ABC flows sharing the ABC router.

4.1 Deployment with non-ABC Routers

An ABC flow can encounter both ABC and non-ABC routers on its path. For example, a Wi-Fi user's traffic may traverse both a Wi-Fi router (running ABC) and an ISP router (not running ABC); either router could be the bottleneck at any given time. ABC flows must therefore be able to detect and react to traditional congestion signals—both drops and ECN—and they must determine when to ignore accelerate feedback from ABC routers because the bottleneck is at a non-ABC router.

We augment the ABC sender to maintain two congestion windows, one for tracking the available rate on ABC routers (w_{abc}), and one for tracking the rate on non-ABC bottlenecks (w_{nonabc}). w_{abc} obeys accelerates/brakes using Equation (3), while w_{nonabc} follows a rule such as

[4] Interestingly, if the sources do not perform additive increase or if the additive increase is sufficiently "gentle," ABC is stable for any value of δ. See the proof in Appendix C for details.

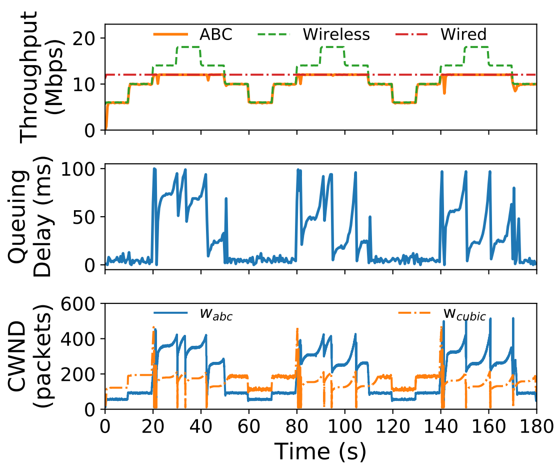


Figure 4: **Coexistence with non-ABC bottlenecks —** When the wired link is the bottleneck, ABC becomes limited by w_{cubic} and behaves like a Cubic flow. When the wireless link is the bottleneck, ABC uses w_{abc} to achieve low delays and high utilization.

Cubic [23] and responds to drop and ECN signals.[5] An ABC sender must send packets to match the lower of the two windows. Our implementation mimics Cubic for the non-ABC method, but other methods could also be emulated.

With this approach, the window that is not the bottleneck could become large. For example, when a non-ABC router is the bottleneck, the ABC router will continually send accelerate signals, causing w_{abc} to grow. If the ABC router later becomes the bottleneck, it will temporarily incur large queues. To prevent this problem, ABC senders cap both w_{abc} and w_{nonabc} to 2× the number of in-flight packets.

Fig. 4 shows the throughput and queuing delay for an ABC flow traversing a path with an ABC-capable wireless link and a wired link with a droptail queue. For illustration, we vary the rate of the wireless link in a series of steps every 5 seconds. Over the experiment, the bottleneck switches between the wired and wireless links several times. ABC is able to adapt its behavior quickly and accurately. Depending on which link is the bottleneck, either w_{nonabc} (i.e., w_{cubic}) or w_{abc} becomes smaller and controls the rate of the flow. When the wireless link is the bottleneck, ABC maintains low queuing delay, whereas the queuing delay exhibits standard Cubic behavior when the wired link is the bottleneck. w_{cubic} does not limit ABC's ability to increase its rate when the wireless link is the bottleneck. At these times (e.g., around the 70 s mark), as soon on w_{abc} increases, the number of in-flight packets and the cap on w_{cubic} increases, and w_{cubic} rises immediately.

4.2 Multiplexing with ECN Bits

IP packets have two ECN-related bits: ECT and CE. These two bits are traditionally interpreted as follows:

ECT	CE	Interpretation
0	0	Non-ECN-Capable Transport
0	1	ECN-Capable Transport ECT(1)
1	0	ECN-Capable Transport ECT(0)
1	1	ECN set

Routers interpret both 01 and 10 to indicate that a flow is ECN-capable, and routers change those bits to 11 to mark a packet with ECN. Upon receiving an ECN mark (11), the receiver sets the *ECN Echo (ECE)* flag to signal congestion to the sender. ABC reinterprets the ECT and CE bits as follows:

ECT	CE	Interpretation
0	0	Non-ECN-Capable Transport
0	1	**Accelerate**
1	0	**Brake**
1	1	ECN set

ABC send all packets with accelerate (01) set, and ABC routers signal brakes by flipping the bits to 10. Both 01 and 10 indicate an ECN-capable transport to ECN-capable legacy routers, which will continue to use (11) to signal congestion.

With this design, receivers must be able to echo both standard ECN signals and accelerates/brakes for ABC. Traditional ECN feedback is signaled using the ECE flag. For ABC feedback, we repurpose the NS (nonce sum) bit, which was originally proposed to ensure ECN feedback integrity [17] but has been reclassified as historic [31] due to lack of deployment. Thus, it appears possible to deploy ABC with only simple modifications to TCP receivers.

Deployment in proxied networks: Cellular networks commonly split TCP connections and deploy proxies at the edge [42, 45]. Here, it is unlikely that any non-ABC router will be the bottleneck and interfere with the accel-brake markings from the ABC router. In this case, deploying ABC may not require any modifications to today's TCP ECN receiver. ABC senders (running on the proxy) can use either 10 or 01 to signal an accelerate, and routers can use 11 to indicate a brake. The TCP receiver can echo this feedback using the ECE flag.

4.3 Non-ABC flows at an ABC Router

ABC flows are potentially at a disadvantage when they share an ABC bottleneck link with non-ABC flows.[6] If the non-ABC flows fill up queues and increase queuing delay, the ABC router will reduce ABC's target rate. To ensure fairness in such scenarios, ABC routers isolate ABC and non-ABC packets in separate queues.

We assume that ABC routers can determine whether a packet belongs to an ABC flow. In some deployment scenarios, this is relatively straightforward. For example, in a cellular network deployment with TCP proxies at the edge of the network [42, 45], the operator can deploy ABC at the proxy, and configure the base station to assume that all traffic from the proxy's IP address uses ABC. Other deployment

[5] We discuss how ABC senders distinguish between accelerate/brake and ECN marks in §4.2.

[6] ABC and non-ABC flows may also share a non-ABC link, but in such cases, ABC flows will behave like Cubic and compete fairly with other traffic.

scenarios may require ABC senders to set a predefined value in a packet field like the IPv6 flow label or the IPv4 IPID.

The ABC router assigns weights to the ABC and non-ABC queues, respectively, and it schedules packets from the queues in proportion to their weights. In addition, ABC's target rate calculation considers only ABC's share of the link capacity (which is governed by the weights). The challenge is to set the weights to ensure that the average throughput of long-running ABC and non-ABC flows is the same, no matter how many flows there are.

Prior explicit control schemes address this problem using the TCP loss-rate equation (XCP) or by estimating the number of flows with Zombie Lists (RCP). Relying on the TCP equation requires a sufficient loss rate and does not handle flows like BBR. RCP's approach does not handle short flows. When one queue has a large number of short flows (and hence a low average throughput), RCP increases the weight of that queue. However, the short flows cannot send faster, so the extra bandwidth is taken by long-running flows in the same queue, which get more throughput than long-running flows in the other queue (see §7.5 for experimental results).

To overcome these drawbacks, a ABC router measures the average rate of the K largest flows in each queue using the Space Saving Algorithm [34], which requires $O(K)$ space. It considers any remaining flow in either queue to be short, and it calculates the total rate of the short flows in each queue by subtracting the rate of the largest K flows from the queue's aggregate throughput. ABC uses these rate measurements to estimate the rate demands of the flows. Using these demands, ABC periodically computes the max-min fair rate allocation for the flows, and it sets the weight of each of the two queues to be equal to the total max-min rate allocation of its component flows. This algorithm ensures that long-running flows in the two queues achieve the same average rate, while accounting for demand-limited short flows.

To estimate the demand of the flows, the ABC router assumes that the demand for the top K flows in each queue is X% higher than the current throughput of the flow, and the aggregate demand for the short flows is the same as their throughput. If a top-K flow is unable to increase its sending rate by X%, its queue's weight will be larger than needed, but any unfairness in weight assignment is bounded by X%. Small values of X limit unfairness but can slow down convergence to fairness; our experiments use $X = 10\%$.

5 Estimating Link Rate

We describe how ABC routers can estimate the link capacity for computing the target rate (§3.1.2). We present a technique for Wi-Fi that leverages the inner workings of the Wi-Fi MAC layer, and we discuss options for cellular networks.

5.1 Wi-Fi

We describe how an 802.11n access point (AP) can estimate the average link rate. For simplicity, we first describe our solution when there is a single user (client) connected to the AP. Next, we describe the multi-user case.

We define *link rate* as the potential throughput of the user (i.e., the MAC address of the Wi-Fi client) if it was backlogged at the AP, i.e., if the user never ran out of packets at the AP. In case the router queue goes empty at the AP, the achieved throughput will be less than the link rate.

Challenges: A strawman would be to estimate the link rate using the physical layer bit rate selected for each transmission, which would depend on the modulation and channel code used for the transmission. Unfortunately, this method will overestimate the link rate as the packet transmission times are governed not only by the bitrate, but also by delays for additional tasks (e.g., channel contention and retransmissions [12]). An alternative approach would be to use the fraction of time that the router queue was backlogged as a proxy for link utilization. However, the Wi-Fi MAC's packet batching confounds this approach. Wi-Fi routers transmit packets (frames) in batches; a new batch is transmitted only after receiving an ACK for the last batch. The AP may accumulate packets while waiting for a link-layer ACK; this queue buildup does not necessarily imply that the link is fully utilized. Thus, accurately measuring the link rate requires a detailed consideration of Wi-Fi's packet transmission protocols.

Understanding batching: In 802.11n, data frames, also known as MAC Protocol Data Units (MPDUs), are transmitted in batches called A-MPDUs (Aggregated MPDUs). The maximum number of frames that can be included in a single batch, M, is negotiated by the receiver and the router. When the user is not backlogged, the router might not have enough data to send a full-sized batch of M frames, but will instead use a smaller batch of size $b < M$. Upon receiving a batch, the receiver responds with a single Block ACK. Thus, at a time t, given a batch size of b frames, a frame size of S bits,[7] and an ACK inter-arrival time (i.e., the time between receptions of consecutive block ACKs) of $T_{IA}(b,t)$, the current dequeue rate, $cr(t)$, may be estimated as

$$cr(t) = \frac{b \cdot S}{T_{IA}(b,t)}. \tag{5}$$

When the user is backlogged and $b = M$, then $cr(t)$ above will be equal to the link capacity. However, if the user is not backlogged and $b < M$, how can the AP estimate the link capacity? Our approach calculates $\hat{T}_{IA}(M,t)$, the estimated ACK inter-arrival time *if the user was backlogged and had sent M frames in the last batch.*

We estimate the link capacity, $\hat{\mu}(t)$, as

$$\hat{\mu}(t) = \frac{M \cdot S}{\hat{T}_{IA}(M,t)}. \tag{6}$$

To accurately estimate $\hat{T}_{IA}(M,t)$, we turn to the relationship between the batch size and ACK inter-arrival time. We can

[7] For simplicity, we assume that all frames are of the same size, though our formulas can be generalized easily for varying frame sizes.

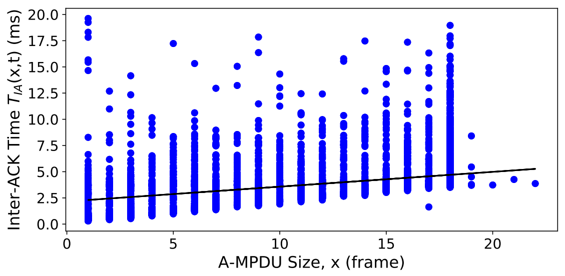


Figure 5: **Inter-ACK time v. batch (A-MPDU) size —** Inter-ACK times for a given batch size exhibits variation. The solid black line represents the average Inter-ACK time. The slope of the line is S/R, where S is the frame size in bits and R is the link rate in bits per second.

decompose the ACK interval time into the batch transmission time and "overhead" time, the latter including physically receiving an ACK, contending for the shared channel, and transmitting the physical layer preamble [21]. Each of these overhead components is independent of the batch size. We denote the overhead time by $h(t)$. If R is the bitrate used for transmission, the router's ACK inter-arrival time is

$$T_{IA}(b,t) = \frac{b \cdot S}{R} + h(t). \tag{7}$$

Fig. 5 illustrates this relationship empirically. There are two key properties to note. First, for a given batch size, the ACK inter-arrival times vary due to overhead tasks. Second, because the overhead time and batch size are independent, connecting the average values of ACK inter-arrival times across all considered batch sizes will produce a line with slope S/R. Using this property along with Equation (7), we can estimate the ACK inter-arrival time for a backlogged user as

$$\begin{aligned}\hat{T_{IA}}(M,t) &= \frac{M \cdot S}{R} + h(t)\\ &= T_{IA}(b,t) + \frac{(M-b) \cdot S}{R}.\end{aligned} \tag{8}$$

We can then use $\hat{T}_{IA}(M,t)$ to estimate the link capacity with Equation (6). This computation is performed for each batch transmission when the batch ACK arrives, and passed through a weighted moving average filter over a sliding window of time T to estimate the smoothed time-varying link rate. T must be greater than the inter-ACK time (up to 20 ms in Fig. 5); we use $T = 40$ ms. Because ABC cannot exceed a rate-doubling per RTT, we cap the predicted link rate to double the current rate (dashed slanted line in Fig. 5).

To evaluate the accuracy of our link rate estimates, we transmit data to a single client through our modified ABC router (§7.1) at multiple different rates over three Wi-Fi links (with different modulation and coding schemes). Fig. 6 summarizes the accuracy of the ABC router's link rate estimates. With this method, the ABC Wi-Fi router is able to predict link rates within 5% of the true link capacities.

Extension to multiple users. In multi-user scenarios, each receiver will negotiate its own maximum batch size (M) with the router, and different users can have different transmission rates. We now present two variants of our technique for (1) when the router uses per-user queues to schedule packets of different users, and (2) when the users share a single FIFO (first-in first-out) queue at the router.

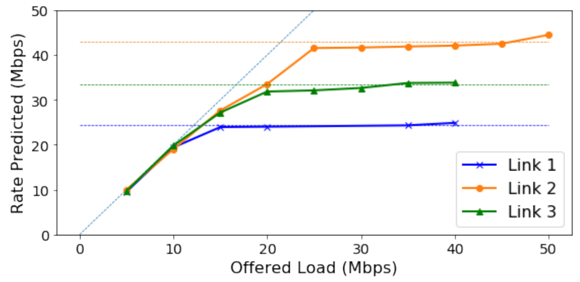


Figure 6: **Wi-Fi Link Rate Prediction —** ABC router link rate predictions for a user that was not backlogged and sent traffic at multiple different rates over three different Wi-Fi links. Horizontal lines represent the true link capacity, solid lines summarize the ABC router's link capacity prediction (each point is an average over 30 seconds of predictions), and the dashed slanted line represents the prediction rate caps. ABC's link rate predictions are within 5% of the ground truth across most sending rates (given the prediction cap).

Per-user queues. In this case each user calculates a separate link rate estimate. Recall that the link rate for a given user is defined as the potential throughput of the user if it was backlogged at the router. To determine the link rate for a user x, we repeat the single-user method for the packets and queue of user x alone, treating transmissions from other users as overhead time. Specifically, user x uses Equations (8) and (6) to compute its link rate ($\hat{\mu}_x(t)$) based on its own values of the bit rate (R_x) and maximum batch size (M_x). It also computes its current dequeue rate ($cr_x(t)$) using Equation (5) to calculate the accel-brake feedback. The inter-ACK time ($T_{IA_x}(b,t)$), is defined as the time between the reception of consecutive block-ACKs for user x. Thus, the overhead time ($h_x(t)$) includes the time when other users at the same AP are scheduled to send packets. Fairness among different users is ensured via scheduling users out of separate queues.

Single queue. In this case the router calculates a single aggregate link rate estimate. The inter-ACK time here is the time between two consecutive block-ACKs, regardless of the user to which the block-ACKs belong to. The router tries to match the aggregate rate of the senders to the aggregate link rate, and uses the aggregate current dequeue rate to calculate accel-brake feedback.

5.2 Cellular Networks

Cellular networks schedule users from separate queues to ensure inter-user fairness. Each user will observe a different link rate and queuing delay. As a result, every user requires a separate target rate calculation at the ABC router. The 3GPP cellular standard [1] describes how scheduling information at the cellular base station can be used to calculate per-user link rates. This method is able to estimate capacity even if a given user is not backlogged at the base station, a key property for the target rate estimation in Equation (1).

6 Discussion

We discuss practical issues pertaining to ABC's deployment.

Delayed Acks: To support delayed ACKs, ABC uses byte counting at the sender; the sender increases/decreases its window by the new bytes ACKed. At the receiver, ABC uses the state machine from DCTCP [6] for generating ACKs and echoing accel/brake marks. The receiver maintains the state of the last packet (accel or brake). Whenever the state changes, the receiver sends an ACK with the new state. If the receiver is in the same state after receiving m packets (the number of ACKs to coalesce), then it sends a delayed ACK with the current state. Our TCP implementation and the experiments in §7 use delayed ACKs with $m = 2$.

Lost ACKs: ABC's window adjustment is robust to ACK losses. Consider a situation where the sender receives a fraction $p < 1$ of the ACKs. If the accelerate fraction at the router is f, the current window of the sender is w_{abc}, then in the next RTT, the change in congestion window of the sender is $fpw_{abc} - (1-f)pw_{abc} = (2f-1)pw_{abc}$. As a result, lost ACKs only slow down the changes in the congestion window, but whether it increases or decreases doesn't depend on p.

ABC routers don't change prior ECN marks: ABC routers don't mark accel-brake on incoming packets that contain ECN marks set by an upstream non-ABC router. Since packets with ECN set can't convey accel-brake marks, they can slow down changes in w_{abc} (similar to lost ACKs). In case the fraction of packets with ECN set is small, then, the slow down in changes to w_{abc} will be small. If the fraction is large, then the non-ABC router is the likely bottleneck, and the sender will not use w_{abc}.

ECN routers clobbering ABC marks: An ECN router can overwrite accel-brake marks. The ABC sender will still track the non-ABC window, w_{nonabc}, but such marks can slow down adjustment to the ABC window, w_{abc}.

ABC on fixed-rate links: ABC can also be deployed on fixed-rate links. On such links, its performance is similar to prior explicit schemes like XCP.

7 Evaluation

We evaluate ABC by considering the following properties:

1. **Performance:** We measure ABC's ability to achieve low delay and high throughput and compare ABC to end-to-end schemes, AQM schemes, and explicit control schemes (§7.3).
2. **Multiple Bottlenecks:** We test ABC in scenarios with multiple ABC bottlenecks and mixtures of ABC and non-ABC bottlenecks (§7.4).
3. **Fairness:** We evaluate ABC's fairness while competing against other ABC and non-ABC flows (§7.5).
4. **Additional Considerations:** We evaluate how ABC performs with application-limited flows and different network delays. We also demonstrate ABC's impact on a real application's performance (§7.6).

7.1 Prototype ABC Implementation

ABC transport: We implemented ABC endpoints in Linux as kernel modules using the pluggable TCP API.

ABC router: We implemented ABC as a Linux queuing discipline (qdisc) kernel module using OpenWrt, an open source operating system for embedded networked devices [18]. We used a NETGEAR WNDR 3800 router configured to 802.11n. We note that our implementation is portable as OpenWrt is supported on many other commodity Wi-Fi routers.

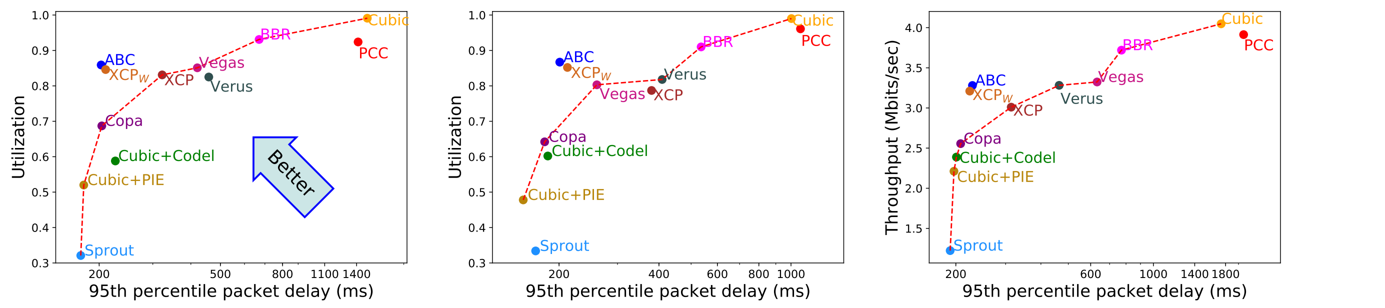
ABC's WiFi link rate estimation exploits the inner workings of the MAC 802.11n protocol, and thus requires fine-grained values at this layer. In particular, the ABC qdisc must know A-MPDU sizes, Block ACK receive times, and packet transmission bitrates. These values are not natively exposed to Linux router qdiscs, and instead are only available at the network driver. To bridge this gap, we modified the router to log the relevant MAC layer data in the cross-layer socket buffer data structure (skb) that it already maintains per packet.

7.2 Experimental Setup

We evaluated ABC in both Wi-Fi and cellular network settings. For Wi-Fi, experiments we used a live Wi-Fi network and the ABC router described in §7.1. For cellular settings, we use Mahimahi [35] to emulate multiple cellular networks (Verizon LTE, AT&T, and TMobile). Mahimahi's emulation uses packet delivery traces (separate for uplink and downlink) that were captured directly on those networks, and thus include outages (highlighting ABC's ability to handle ACK losses).

We compare ABC to end-to-end protocols designed for cellular networks (Sprout [46] and Verus [50]), loss-based end-to-end protocols both with and without AQM (Cubic [23], Cubic+Codel [36], and Cubic+PIE [37]), recently-proposed end-to-end protocols (BBR [14], Copa [10], PCC Vivace-Latency (referred as PCC)) [16]), and TCP Vegas [13]), and explicit control protocols (XCP [29], RCP [43] and VCP [47]). We used TCP kernel modules for ABC, BBR, Cubic, PCC, and Vegas; for these schemes, we generated traffic using iperf [44]. For the end-to-end schemes that are not implemented as TCP kernel modules (i.e., Copa, Sprout, Verus), we used the UDP implementations provided by the authors. Lastly, for the explicit control protocols (i.e., XCP, RCP, and VCP), we used our own implementations as qdiscs with Mahimahi to ensure compatibility with our emulation setup. We used Mahimahi's support of Codel and Pie to evaluate AQM.

Our emulated cellular network experiments used a minimum RTT of 100 ms and a buffer size of 250 MTU-sized packets. Additionally, ABC's target rate calculation (Equation (1)) used $\eta = 0.98$ and $\delta = 133$ ms. Our Wi-Fi implementation uses the link rate estimator from §5, while our emulated cellular network setup assumes the realistic scenario that ABC's router has knowledge of the underlying link capacity [1].



(a) Downlink (b) Uplink (c) Uplink+Downlink

Figure 7: **ABC vs. previous schemes on three Verizon cellular network traces —** In each case, ABC outperforms all other schemes and sits well outside the Pareto frontier of previous schemes (denoted by the dashed lines).

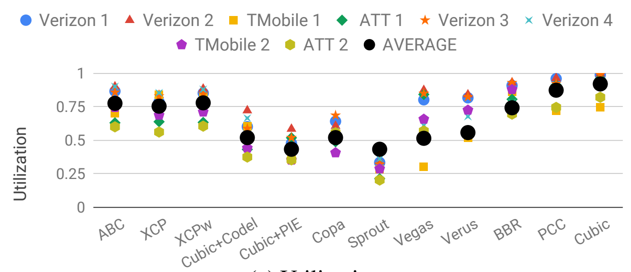
7.3 Performance

Cellular: Fig. 7a and 7b show the utilization and 95^{th} percentile per packet delay that a single backlogged flow achieves using each aforementioned scheme on two Verizon LTE cellular link traces. ABC exhibits a better (i.e., higher) throughput/delay tradeoff than all prior schemes. In particular, ABC sits well outside the Pareto frontier of the existing schemes, which represents the prior schemes that achieve higher throughput or lower delay than any other prior schemes.

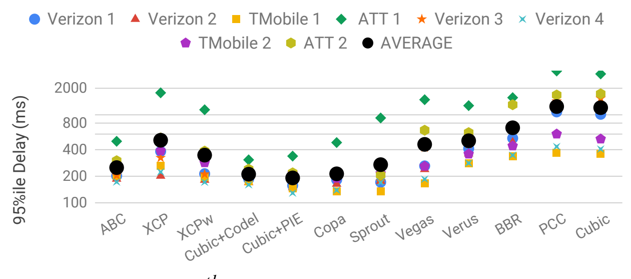
Further analysis of Fig. 7a and 7b reveals that Cubic+Codel, Cubic+PIE, Copa, and Sprout are all able to achieve low delays that are comparable to ABC. However, these schemes heavily underutilize the link. The reason is that, though these schemes are able to infer and react to queue buildups in a way that reduces delays, they lack a way of quickly inferring increases in link capacity (a common occurrence on time-varying wireless links), leading to underutilization. In contrast, schemes like BBR, Cubic, and PCC are able to rapidly saturate the network (achieving high utilization), but these schemes also quickly fill buffers and thus suffer from high queuing delays. Unlike these prior schemes, ABC is able to quickly react to *both* increases and decreases in available link capacity, enabling high throughput and low delays.

We observed similar trends across a larger set of 8 different cellular network traces (Fig. 8). ABC achieves 50% higher throughput than Cubic+Codel and Copa, while only incurring 17% higher 95^{th} percentile packet delays. PCC and Cubic achieve slightly higher link utilization values than ABC (12%, and 18%, respectively), but incur significantly higher per-packet delays than ABC (394%, and 382%, respectively). Finally, compared to BBR, Verus, and Sprout, ABC achieves higher link utilization (4%, 39%, and 79%, respectively). BBR and Verus incur higher delays (183% and 100%, respectively) than ABC. Appendix E shows mean packet delay over the same conditions, and shows the same trends.

Comparison with Explicit Protocols: Fig. 7 and 8 also show that ABC outperforms the explicit control protocol, XCP, despite not using multi-bit per-packet feedback as XCP does. For XCP we used $\alpha = 0.55$ and $\beta = 0.4$, the highest permissible stable values that achieve the fastest possible link rate convergence. XCP achieves similar average throughput to ABC, but with 105% higher 95^{th} percentile delays. This performance discrepancy can be attributed to the fact that ABC's control rule is better suited for the link rate variations in wireless networks. In particular, unlike ABC which updates its feedback on every packet, XCP computes aggregate feedback values (ϕ) only once per RTT and may thus take an entire RTT to inform a sender to reduce its window. To overcome this, we also considered an improved version of XCP that recomputes aggregate feedback on each packet based on the rate and delay measurements from the past RTT; we refer to this version as XCP_w (short for XCP wireless). As shown in Fig. 7 and Fig. 8, XCP_w reduces delay compared to XCP, but still incurs 40% higher 95^{th} percentile delays (averaged across traces) than ABC. We also compared with two other explicit schemes, RCP and VCP, and found that ABC consistently outperformed both, achieving 20% more utilization on average. (Appendix F).

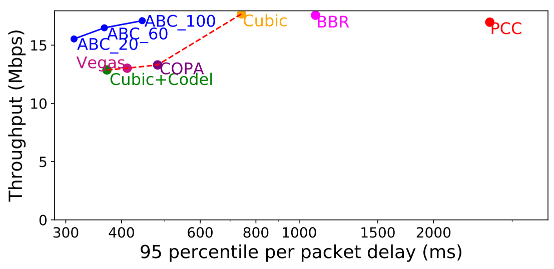


(a) Utilization

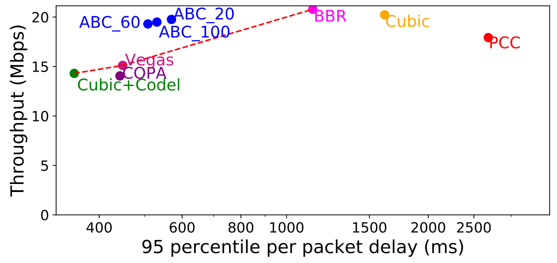


(b) 95^{th} percentile per-packet delay

Figure 8: **95^{th} percentile per-packet delay across 8 cellular link traces —** On average, ABC achieves similar delays and 50% higher utilization than Copa and Cubic+Codel. PCC and Cubic achieve slightly higher throughput than ABC, but incur 380% higher 95^{th} percentile delay than ABC.



(a) Single user



(b) Two users, shared queue

Figure 9: **Throughout and mean delay on Wi-Fi —** For the multi-user scenario, we report the sum of achieved throughputs and the average of observed 95th percentile delay across both users. We consider three versions of ABC (denoted ABC _*) for different delay thresholds. All versions of ABC outperform all prior schemes and sit outside the pareto frontier.

Wi-Fi: We performed similar evaluations on a live Wi-Fi link, considering both single and multi-user scenarios. We connect senders to a WiFi router via Ethernet. Each sender transmits data through the WiFi router to one receiver. All receivers' packets share the same FIFO queue at the router. In this experiment, we excluded Verus and Sprout as they are designed specifically for cellular networks. To mimic common Wi-Fi usage scenarios where endpoints can move and create variations in signal-to-noise ratios (and thus bitrates), we varied the Wi-Fi router's bitrate selections by varying the MCS index using the Linux `iw` utility; we alternated the MCS index between values of 1 and 7 every 2 seconds. In Appendix 16, we also list results for an experiment where we model MCS index variations as Brownian motion—results show the same trends as described below. This experiment was performed in a crowded computer lab with contention from other Wi-Fi networks. We report average performance values across three, 45 second runs. We considered three different ABC delay threshold (d_t) values of 20 ms, 60 ms, and 100 ms; note that increasing ABC's delay threshold will increase both observed throughput and RTT values.

Fig. 9 shows the throughput and 95th percentile per-packet delay for each protocol. For the multi-user scenario, we report the sum of achieved throughputs and the average 95th percentile delay across all users. In both the single and multi-user scenarios, ABC achieves a better throughput/delay tradeoff than all prior schemes, and falls well outside the Pareto frontier for those schemes. In the single user scenario, the ABC configuration with $d_t = 100$ ms achieves up to 29% higher throughput than Cubic+Codel, Copa and Vegas. Though PCC-Vivace, Cubic and BBR achieve slightly higher throughput (4%) than this ABC configuration, their delay values are considerably higher (67%-6×). The multi-user scenario showed similar results. For instance, ABC achieves 38%, 41% and 31% higher average throughput than Cubic+Codel, Copa and Vegas, respectively.

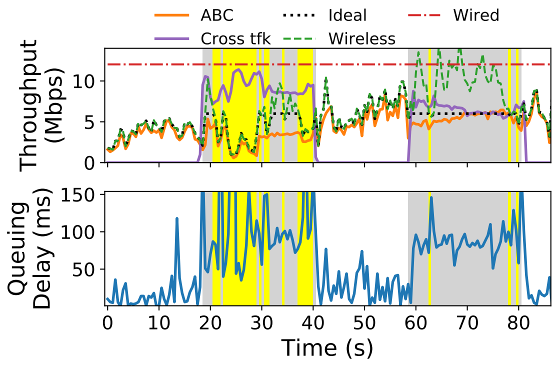


Figure 10: **Coexistence with non-ABC bottlenecks —** ABC tracks the ideal rate closely (fair share) and reduces queuing delays in the absence of cross traffic (white region).

7.4 Coexistence with Various Bottlenecks

Coexistence with ABC bottlenecks: Fig. 7c compares ABC and prior protocols on a network path with two cellular links. In this scenario, ABC tracks the bottleneck link rate and achieves a better throughput/delay tradeoff than prior schemes, and again sits well outside the Pareto frontier.

Coexistence with non-ABC bottlenecks: Fig. 10 illustrates throughput and queuing delay values for an ABC flow traversing a network path with both an emulated wireless link and an emulated 12 Mbits/s fixed rate (wired) link. The wireless link runs ABC, while the wired link operates a droptail buffer. ABC shares the wired link with on-off cubic cross traffic. In the absence of cross traffic (white region), the wireless link is always the bottleneck. However, with cross traffic (yellow and grey regions), due to contention, the wired link can become the bottleneck. In this case, ABC's fair share on the wired link is half of the link's capacity (i.e., 6 Mbit/s). If the wireless link rate is lower than the fair share on the wired link (yellow region), the wireless link remains the bottleneck; otherwise, the wired link becomes the bottleneck (grey region).

The black dashed line in the top graph represents the ideal fair throughput for the ABC flow throughout the experiment. As shown, in all regions, ABC is able to track the ideal rate closely, even as the bottleneck shifts. In the absence of cross traffic, ABC achieves low delays while maintaining high link utilization. With cross traffic, ABC appropriately tracks the wireless link rate (yellow region) or achieves its fair share of the wired link (grey region) like Cubic. In the former cross traffic scenario, increased queuing delays are due to congestion caused by the Cubic flow on the wired link. Further, deviations from the ideal rate in the latter cross traffic scenario can

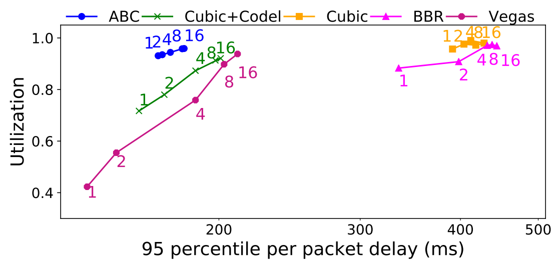


Figure 11: **Coexistence among ABC flows —** ABC achieves similar aggregate utilization and delay irrespective of the number of connections. ABC outperforms all previous schemes.

be attributed to the fact that the ABC flow is running as Cubic, which in itself takes time to converge to the fair share [23].

7.5 Fairness among ABC and non-ABC flows

Coexistence among ABC flows: We simultaneously run multiple ABC flows on a fixed 24 Mbits/s link. We varied the number of competing flows from 2 to 32 (each run was 60 s). In each case, the Jain Fairness Index [28] was within 5% from the ideal fairness value of 1, highlighting ABC's ability to ensure fairness.

Fig. 11 shows the aggregate utilization and delay for concurrent flows (all flows running the same scheme) competing on a Verizon cellular link. We varied the number of competing flows from 1 to 16. ABC achieves similar aggregate utilization and delay across all scenarios, and, outperforms all other schemes. For all the schemes, the utilization and delay increase when the number of flows increases. For ABC, this increase can be attributed to the additional packets that result from additive increase (1 packet per RTT per flow). For other schemes, this increase is because multiple flows in aggregate ramp-up their rates faster than a single flow.

RTT Unfairness: We simultaneously ran 2 ABC flows on a 24 Mbits wired bottleneck. We varied the RTT of flow 1 from 20ms to 120ms. RTT of flow 2 was fixed to 20ms. Fig. 12 shows the ratio of the average throughput of these 2 flows (average throughput of flow 2 / flow 1, across 5 runs) against the ratio of their RTTs (RTT of flow 1 / flow 2). Increasing the RTT ratio increases the throughput ratio almost linearly and the throughput is inversely proportional to the RTT. Thus, the unfairness is similar to existing protocols like Cubic.

Next, we simultaneously ran 6 ABC flows. The RTT of the flows vary from 20ms to 120ms. Table 1 shows the RTT and the average throughput across 5 runs. Flows with higher RTTs have lower throughput. However, note that the flow with the highest RTT (120ms) still achieves ∼35 % of the throughput as flow with the lowest RTT (20ms).

Coexistence with non-ABC flows: We consider a scenario where 3 ABC and 3 non-ABC (in this case, Cubic) long-lived flows share the same 96 Mbits/s bottleneck link. In addition, we create varying numbers of non-ABC short flows (each of size 10 KB) with Poisson flow arrival times to offer a fixed average load. We vary the offered load values, and report

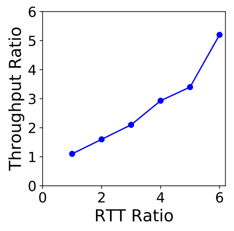
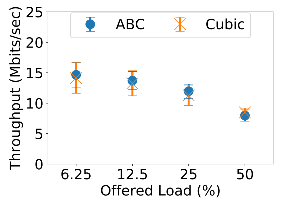


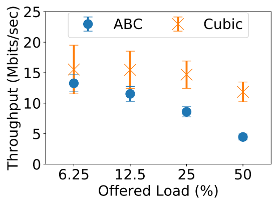
Figure 12: **RTT unfairness**

RTT (ms)	Tput (Mbps)
20	6.62
40	4.94
60	4.27
80	3.0
100	2.75
120	2.40

Table 1: **RTT unfairness**



(a) ABC



(b) RCP's Zombie List

Figure 13: **Coexistence with non-ABC flows —** Across all scenarios, the standard deviation for ABC flows is small and the flows are fair to each other. Compared to RCP's Zombie List strategy, ABC's max-min allocation provides better fairness between ABC and non-ABC flows. With ABC's strategy, the difference in average throughput of ABC and Cubic flows is under 5%.

results across 10 runs (40 seconds each). We compare ABC's strategy to coexist with non-ABC flows to RPC's Zombie list approach (§4.3).

Fig. 13 shows the mean and standard deviation of throughput for long-lived ABC and Cubic flows. As shown in Fig. 13a, ABC's coexistence strategy allows ABC and Cubic flows to fairly share the bottleneck link across all offered load values. Specifically, the difference in average throughput between the ABC and Cubic flows is under 5%. In contrast, Fig. 13b shows that RCP's coexistence strategy gives higher priority to Cubic flows. This discrepancy increases as the offered load increases, with Cubic flows achieving 17-165% higher throughput than ABC flows. The reason, as discussed in §4.3, is that long-lived Cubic flows receive higher throughput than the average throughput that RCP estimates for Cubic flows. This leads to unfairness because RCP attempts to match average throughput for each scheme.Fig. 13 also shows that the standard deviation of ABC flows is small (under 10%) across all scenarios. This implies that in each run of the experiment, the throughput for each of the three concurrent ABC flows is close to each other, implying fairness across ABC flows. Importantly, the standard deviation values for ABC are smaller than those for Cubic. Thus, ABC flows converge to fairness faster than Cubic flows do.

7.6 Additional Results

ABC's sensitivity to network latency: Thus far, our emulation experiments have considered fixed minimum RTT values of 100 ms. To evaluate the impact that propagation delay on ABC's performance, we repeated the experiment from Fig. 8 on the RTT values of 20 ms, 50 ms, 100 ms, and 200 ms. Across all RTTs, ABC outperforms all prior schemes, again achieving a more desirable throughput/latency trade off (see Appendix G).

Application-limited flows: We created a single long-lived ABC flow that shared a cellular link with 200 application-limited ABC flows that send traffic at an aggregate of 1 Mbit/s. Despite the fact that the application-limited flows do not have traffic to properly respond to ABC's feedback, the ABC flows (in aggregate) still achieve low queuing delays and high link utilization. See Appendix G for details.

Perfect future capacity knowledge: We considered a variant of ABC, PK-ABC, which knows an entire emulated link trace in advance. This experiment reflects the possibility of resource allocation predictions at cellular base stations. Rather than using an estimate of the current link rate to compute a target rate (as ABC does), PK-ABC uses the expected link rate 1 RTT in the future. On the same setup as Fig. 7b, PK-ABC reduces 95^{th} percentile per-packet-delays from 97 ms to 28 ms, compared to ABC, while achieving similar utilization (~90%).

ABC's improvement on real applications: We evaluated ABC's improvement for real user-facing applications on a multiplayer interactive game, Slither.io [3]. We loaded Slither.io using a Google Chrome browser which ran inside an emulated cellular link with a background backlogged flow. We considered three schemes for the backlogged flow: Cubic, Cubic+Codel, and ABC. Cubic fully utilizes the link, but adds excessive queuing delays hindering gameplay. Cubic+Codel reduces queuing delays (improving user experience in the game), but underutilizes the link. Only ABC is able to achieve both high link utilization for the backlogged flow and low queuing delays for the game. A video demo of this experiment can be viewed at `https://youtu.be/Dauq-tfJmyU`.

Impact of η: This parameter presents a trade-off between throughput and delay. Increasing η increases the throughput but at the cost of additional delay (see Appendix B).

8 Related Work

Several prior works have proposed using LTE infrastructure to infer the underlying link capacity [26, 33, 48]. CQIC [33] and piStream [48] use physical layer information at the receiver to estimate link capacity. However, these approaches have several limitations that lead to inaccurate estimates. CQIC's estimation approach considers historical resource usage (not the available physical resources) [48], while piStream's technique relies on second-level video segment downloads and thus does not account for the short timescale variations in link rate required for per-packet congestion control. These inaccuracies stem from the opacity of the base station's resource allocation process at the receiver. ABC circumvents these issues by accurately estimating link capacity directly at the base station.

In VCP [47], router classifies congestion as low, medium, or high, and signals the sender to either perform a multiplicative increase, additive increase, or multiplicative decrease in response. Unlike an ABC sender, which reacts to ACKs individually, VCP senders act once per RTT. This coarse-grained update limits VCP's effectiveness on time-varying wireless paths. For instance, it can take 12 RTTs to double the window. VCP is also incompatible with ECN, making it difficult to deploy.

In BMCC [39, 40], a router uses ADPM [9] to send link load information to the receiver on ECN bits, relying on TCP options to relay the feedback from the receiver to the sender. MTG proposed modifying cellular base stations to communicate the link rate explicitly using a new TCP option [26]. Both approaches do not work with IPSec encryption [30], and such packet modifications trigger the risk of packets being dropped silently by middleboxes [25]. Moreover, unlike ABC, MTG does not ensure fairness among multiple flows for a user, while BMCC has the same problem with non-BMCC flows [38, 39].

XCP-b [4] is a variant of XCP designed for wireless links with unknown capacity. XCP-b routers use the queue size to determine the feedback. When the queue is backlogged, the XCP-b router calculates spare capacity using the change in queue size and uses the same control rule as XCP. When the queue goes to zero, XCP-b cannot estimate spare capacity, and resorts to a blind fixed additive increase. Such blind increase can cause both under-utilization and increased delays (§2.)

Although several prior schemes (XCP, RCP, VCP, BMCC, XCP-b) attempt to match the current enqueue rate to the capacity, none match the future enqueue rate to the capacity, and so do not perform as well as ABC on time-varying links.

9 Conclusion

This paper presented a simple new explicit congestion control protocol for time-varying wireless links called ABC. ABC routers use a single bit to mark each packet with "accelerate" or "brake", which causes senders to slightly increase or decrease their congestion windows. Routers use this succinct feedback to quickly guide senders towards a desired target rate. ABC outperforms the best existing explicit flow control scheme, XCP, but unlike XCP, ABC does not require modifications to packet formats or user devices, making it simpler to deploy. ABC is also incrementally deployable: ABC can operate correctly with multiple ABC and non-ABC bottlenecks, and can fairly coexist with ABC and non-ABC traffic sharing the same bottleneck link. We evaluated ABC using a WiFi router implementation and trace-driven emulation of cellular links. ABC achieves 30-40% higher throughput than Cubic+Codel for similar delays, and 2.2× lower delays than BBR on a Wi-Fi path. On cellular network paths, ABC achieves 50% higher throughput than Cubic+Codel.

Acknowledgments. We thank Anirudh Sivaraman, Peter Iannucci, and Srinivas Narayana for useful discussions. We are grateful to the anonymous reviewers and our shepherd Jitendra Padhye for their feedback and useful comments. This work was supported in part by DARPA under Contract No. HR001117C0048, and NSF grants CNS-1407470, CNS-1751009, and CNS-1617702.

References

[1] 3GPP technical specification for lte. https://www.etsi.org/deliver/etsi_ts/132400_132499/132450/09.01.00_60/ts_132450v090100p.pdf.

[2] sfqCoDel. http://www.pollere.net/Txtdocs/sfqcodel.cc.

[3] Slither.io interactive multiplayer game. http://slither.io.

[4] F. Abrantes and M. Ricardo. XCP for shared-access multi-rate media. *Computer Communication Review*, 36(3):27–38, 2006.

[5] A. Akella, S. Seshan, S. Shenker, and I. Stoica. Exploring congestion control. Technical report, CMU School of Computer Science, 2002.

[6] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In S. Kalyanaraman, V. N. Padmanabhan, K. K. Ramakrishnan, R. Shorey, and G. M. Voelker, editors, *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New Delhi, India, August 30 -September 3, 2010*, pages 63–74. ACM, 2010.

[7] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In S. D. Gribble and D. Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 253–266. USENIX Association, 2012.

[8] M. Allman. Tcp congestion control with appropriate byte counting (abc)", rfc 3465. 2003.

[9] L. L. H. Andrew, S. V. Hanly, S. Chan, and T. Cui. Adaptive deterministic packet marking. *IEEE Communications Letters*, 10(11):790–792, 2006.

[10] V. Arun and H. Balakrishnan. Copa: Practical delay-based congestion control for the internet. In S. Banerjee and S. Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 329–342. USENIX Association, 2018.

[11] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic behavior of slowly-responsive congestion control algorithms. In R. L. Cruz and G. Varghese, editors, *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA*, pages 263–274. ACM, 2001.

[12] J. C. Bicket. *Bit-rate selection in wireless networks*. PhD thesis, Massachusetts Institute of Technology, 2005.

[13] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In J. Crowcroft, editor, *Proceedings of the ACM SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications, London, UK, August 31 - September 2, 1994*, pages 24–35. ACM, 1994.

[14] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: congestion-based congestion control. *ACM Queue*, 14(5):20–53, 2016.

[15] D. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks*, 17:1–14, 1989.

[16] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira. PCC vivace: Online-learning congestion control. In S. Banerjee and S. Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 343–356. USENIX Association, 2018.

[17] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust congestion signaling. In *Network Protocols, 2001. Ninth International Conference on*, pages 332–341. IEEE, 2001.

[18] F. Fainelli. The openwrt embedded development framework. In *Proceedings of the Free and Open Source Software Developers European Meeting*, 2008.

[19] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.

[20] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. Ip options are not an option. *University of California at Berkeley, Technical Report UCB/EECS-2005-24*, 2005.

[21] B. Ginzburg and A. Kesselman. Performance analysis of a-mpdu and a-msdu aggregation in ieee 802.11 n. In *Sarnoff symposium, 2007 IEEE*, pages 1–5. IEEE, 2007.

[22] P. Goyal, M. Alizadeh, and H. Balakrishnan. Rethinking congestion control for cellular networks. In S. Banerjee, B. Karp, and M. Walfish, editors, *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, pages 29–35. ACM, 2017.

[23] S. Ha, I. Rhee, and L. Xu. CUBIC: a new tcp-friendly high-speed TCP variant. *Operating Systems Review*, 42(5):64–74, 2008.

[24] J. C. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In D. Estrin, S. Floyd, C. Partridge, and M. Steenstrup, editors, *Proceedings of the ACM SIGCOMM 1996 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Stanford, CA, USA, August 26-30, 1996*, pages 270–280. ACM, 1996.

[25] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In P. Thiran and W. Willinger, editors, *Proceedings of the 11th ACM SIGCOMM Internet Measurement Conference, IMC '11, Berlin, Germany, November 2-, 2011*, pages 181–194. ACM, 2011.

[26] A. Jain, A. Terzis, H. Flinck, N. Sprecher, S. Arunachalam, and K. Smith. Mobile throughput guidance inband signaling protocol. *IETF, work in progress*, 2015.

[27] R. Jain. Congestion control and traffic management in ATM networks: Recent advances and a survey. *Computer Networks and ISDN Systems*, 28(13):1723–1738, 1996.

[28] R. Jain, A. Durresi, and G. Babic. Throughput fairness index: An explanation. In *ATM Forum contribution*, volume 99, 1999.

[29] D. Katabi, M. Handley, and C. E. Rohrs. Congestion control for high bandwidth-delay product networks. In M. Mathis, P. Steenkiste, H. Balakrishnan, and V. Paxson, editors, *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 19-23, 2002, Pittsburgh, PA, USA*, pages 89–102. ACM, 2002.

[30] S. T. Kent and K. Seo. Security architecture for the internet protocol. *RFC*, 4301:1–101, 2005.

[31] M. Kühlewind and R. Scheffenegger. Design and evaluation of schemes for more accurate ECN feedback. In *Proceedings of IEEE International Conference on Communications, ICC 2012, Ottawa, ON, Canada, June 10-15, 2012*, pages 6937–6941. IEEE, 2012.

[32] S. S. Kunniyur and R. Srikant. Analysis and design of an adaptive virtual queue (AVQ) algorithm for active queue management. In R. L. Cruz and G. Varghese, editors, *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA*, pages 123–134. ACM, 2001.

[33] F. Lu, H. Du, A. Jain, G. M. Voelker, A. C. Snoeren, and A. Terzis. CQIC: revisiting cross-layer congestion control for cellular networks. In J. Manweiler and R. R. Choudhury, editors, *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications, HotMobile 2015, Santa Fe, NM, USA, February 12-13, 2015*, pages 45–50. ACM, 2015.

[34] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In T. Eiter and L. Libkin, editors, *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, volume 3363 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[35] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In S. Lu and E. Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 417–429. USENIX Association, 2015.

[36] K. M. Nichols and V. Jacobson. Controlling queue delay. *Commun. ACM*, 55(7):42–50, 2012.

[37] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *IEEE 14th International Conference on High Performance Switching and Routing, HPSR 2013, Taipei, Taiwan, July 8-11, 2013*, pages 148–155. IEEE, 2013.

[38] I. A. Qazi, L. Andrew, and T. Znati. Incremental deployment of new ecn-compatible congestion control. In *Proc. PFLDNeT*, 2009.

[39] I. A. Qazi, L. L. H. Andrew, and T. Znati. Congestion control with multipacket feedback. *IEEE/ACM Trans. Netw.*, 20(6):1721–1733, 2012.

[40] I. A. Qazi, T. Znati, and L. L. H. Andrew. Congestion control using efficient explicit feedback. In *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*, pages 10–18. IEEE, 2009.

[41] K. K. Ramakrishnan, S. Floyd, and D. L. Black. The addition of explicit congestion notification (ECN) to IP. *RFC*, 3168:1–63, 2001.

[42] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: controlling user-perceived delays in server-based mobile applications. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington,*

PA, USA, November 3-6, 2013, pages 85–100. ACM, 2013.

[43] C. Tai, J. Zhu, and N. Dukkipati. Making large scale deployment of RCP practical for real networks. In *INFOCOM 2008. 27th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 13-18 April 2008, Phoenix, AZ, USA*, pages 2180–2188. IEEE, 2008.

[44] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. `http://dast.nlanr.net/Projects`, 2005.

[45] Z. Wang, Z. Qian, Q. Xu, Z. M. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In S. Keshav, J. Liebeherr, J. W. Byers, and J. C. Mogul, editors, *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 374–385. ACM, 2011.

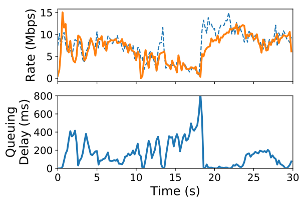
[46] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In N. Feamster and J. C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 459–471. USENIX Association, 2013.

[47] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman. One more bit is enough. In R. Guérin, R. Govindan, and G. Minshall, editors, *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, pages 37–48. ACM, 2005.

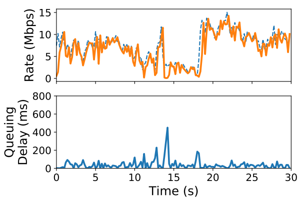
[48] X. Xie, X. Zhang, S. Kumar, and L. E. Li. pistream: Physical layer informed adaptive video streaming over LTE. In S. Fdida, G. Pau, S. K. Kasera, and H. Zheng, editors, *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, MobiCom 2015, Paris, France, September 7-11, 2015*, pages 413–425. ACM, 2015.

[49] J. A. Yorke. Asymptotic stability for one dimensional differential-delay equations. *Journal of Differential equations*, 7(1):189–202, 1970.

[50] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg. Adaptive congestion control for unpredictable cellular networks. In S. Uhlig, O. Maennel, B. Karp, and J. Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 509–522. ACM, 2015.



(a) BBR



(b) ABC

Figure 14: **Comparison with BBR —** BBR overshoots the link capacity, causing excessive queuing. Same setup as Fig. 1.

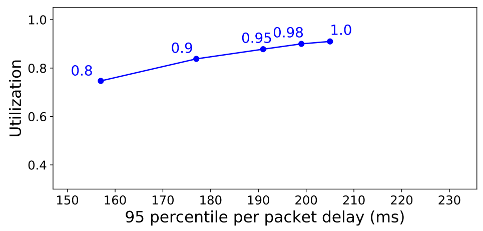


Figure 15: **Impact of** η **—** Performance of ABC with various values of η (target utilization). η presents a trade-off between throughput and delay. Same setup as Fig. 1.

A BBR overestimates the sending rate

Fig. 14 shows the throughput and queuing delay of BBR on a Verizon cellular trace. BBR periodically increases its rate in short pulses, and frequently overshoots the link capacity with variable-bandwidth links, causing excessive queuing.

B Impact of η

Fig. 15 shows the performance of ABC with various values of η on a Verizon cellular trace. Increasing η increases the link utilization, but, also increases the delay. Thus, η presents a trade-off between throughput and delay.

C Stability Analysis

This section establishes the stability bounds for ABC's control algorithm (Theorem 1).

Model: Consider a single ABC link, traversed by N ABC flows. Let $\mu(t)$ be the link capacity at time t. As $\mu(t)$ can be time-varying, we define stability as follows. Suppose that at some time t_0, $\mu(t)$ stops changing, i.e., for $t > t_0$ $\mu(t) = \mu$ for some constant μ. We aim to derive conditions on ABC's parameters which guarantee that the aggregate rate of the senders and the queue size at the routers will converge to certain fixed-point values (to be determined) as $t \to \infty$.

Let τ be the common round-trip propagation delay on the path for all users. For additive increase (§4.3), assume that each sender increases its congestion window by 1 every l seconds. Let $f(t)$ be the fraction of packets marked accelerate, and, $cr(t)$ be the dequeue rate at the ABC router at time t. Let τ_r be time it takes accel-brake marks leaving the ABC router to reach the sender. Assuming that there are no queues other than at the ABC router, τ_r will be the sum of the propagation delay between ABC router and the receiver and the propagation delay between receiver and the senders. The aggregate incoming rate of ACKs across all the senders at time t, $R(t)$, will be equal to the dequeue rate at the router at time $t - \tau_r$:

$$R(t) = cr(t - \tau_r). \tag{9}$$

In response to an accelerate, a sender will send 2 packets, and, for a brake, a sender won't send anything. In addition to responding to accel-brakes, each sender will also send an additional packet every l seconds (because of AI). Therefore, the aggregate sending rate for all senders at time t, $S(t)$, will be

$$\begin{aligned} S(t) &= R(t) \cdot 2 \cdot f(t - \tau_r) + \frac{N}{l} \\ &= 2cr(t - \tau_r) f(t - \tau_r) + \frac{N}{l}. \end{aligned} \tag{10}$$

Substituting $f(t - \tau_r)$ from Equation (2), we get

$$S(t) = tr(t - \tau_r) + \frac{N}{l}. \tag{11}$$

Let τ_f be the propagation delay between a sender and the ABC router, and $eq(t)$ be the enqueue rate at the router at time t. Then $eq(t)$ is given by

$$\begin{aligned} eq(t) &= S(t - \tau_f) \\ &= tr(t - (\tau_r + \tau_f)) + \frac{N}{l} \\ &= tr(t - \tau) + \frac{N}{l}. \end{aligned} \tag{12}$$

Here, $\tau = \tau_r + \tau_f$ is the round-trip propagation delay.

Let $q(t)$ be the queue size, and, $x(t)$ be the queuing delay at time t:

$$x(t) = \frac{q(t)}{\mu}.$$

Ignoring the boundary conditions for simplicity ($q(t)$ must be ≥ 0), the queue length has the following dynamics:

$$\begin{aligned} \dot{q}(t) &= eq(t) - \mu \\ &= tr(t - \tau) + \frac{N}{l} - \mu \\ &= \left((\eta - 1) \cdot \mu + \frac{N}{l} \right) - \frac{\mu}{\delta} (x(t - \tau) - d_t)^+, \end{aligned}$$

where in the last step we have used Equation (1). Therefore the dynamics of $x(t)$ can be described by:

$$\begin{aligned} \dot{x}(t) &= \left((\eta - 1) + \frac{N}{\mu \cdot l} \right) - \frac{1}{\delta} (x(t - \tau) - d_t)^+ \\ &= A - \frac{1}{\delta} (x(t - \tau) - d_t)^+, \end{aligned} \tag{13}$$

where $A = \left((\eta - 1) + \frac{N}{\mu \cdot l} \right)$, and, A is a constant given a fixed number of flows N. The delay-differential equation in Equation (13) captures the behavior of the entire system. We use it to analyze the behavior of the queuing delay, $x(t)$, which in turn informs the dynamics of the target rate, $tr(t)$, and enqueue rate, $eq(t)$, using Equations (1) and (12) respectively.

Stability: For stability, we consider two possible scenarios 1) $A<0$, and 2) $A \geq 0$. We argue the stability in each case.

Case 1: $A<0$. In this case, the stability analysis is straightforward. The fixed point for queuing delay, x^*, is 0. From Equation (13), we get

$$\dot{x}(t) = A - \frac{1}{\delta}(x(t-\tau)-d_t)^+ \leq A < 0. \tag{14}$$

The above equation implies that the queue delay will decrease at least as fast as A. Thus, the queue will go empty in a bounded amount of time. Once the queue is empty, it will remain empty forever, and the enqueue rate will converge to a fixed value. Using Equation (12), the enqueue rate can will converge to

$$\begin{aligned} eq(t) &= tr(t-\tau)+\frac{N}{l} \\ &= \eta\mu+\frac{N}{l}-\frac{\mu}{\delta}(x(t-\tau)-d_t)^+ \\ &= \eta\mu+\frac{N}{l} \\ &= (1+A)\mu. \end{aligned} \tag{15}$$

Note that $\eta\mu<(1+A)\mu<\mu$. Since both the enqueue rate and the queuing delay converge to fixed values, the system is stable for any value of δ.

Case 2: $A>0$: The fixed point for the queuing delay in this case is $x^* = A\cdot\delta + d_t$. Let $\tilde{x}(t) = x(t) - x^*$ be the deviation of the queuing delay from its fixed point. Substituting in Equation (13), we get

$$\begin{aligned} \dot{\tilde{x}}(t) &= A - \frac{1}{\delta}(\tilde{x}(t-\tau)+A\cdot\delta)^+ \\ &= -max(-A, \frac{1}{\delta}\tilde{x}(t-\tau)) \\ &= -g(\tilde{x}(t-\tau)), \end{aligned} \tag{16}$$

where $g(u) = \max(-A, \frac{1}{\delta}u)$ and $A>0$.

In [49] (Corollary 3.1), Yorke established that delay-differential equations of this type are globally asymptotically stable (i.e., $\tilde{x}(t) \to 0$ as $t \to \infty$ irrespective of the initial condition), if the following conditions are met:

1. $\mathbf{H}_1$**:** g is continuous.
2. $\mathbf{H}_2$**:** There exists some α, s.t. $\alpha\cdot u^2 > ug(u) > 0$ for all $u \neq 0$.
3. $\mathbf{H}_3$**:** $\alpha\cdot\tau < \frac{3}{2}$.

The function $g(\cdot)$ trivially satisfies $\mathbf{H}_1$. $\mathbf{H}_2$ holds for any $\alpha \in (\frac{1}{\delta}, \infty)$. Therefore, there exists an $\alpha \in (\frac{1}{\delta}, \infty)$ that satisfies both $\mathbf{H}_2$ and $\mathbf{H}_3$ if

$$\frac{1}{\delta}\cdot\tau < \frac{3}{2} \implies \delta > \frac{2}{3}\cdot\tau. \tag{17}$$

This proves that ABC's control rule is asymptotically stable if Equation (17) holds. Having established that $x(t)$ converges to $x^* = A\cdot\delta + d_t$, we can again use Equation (12) to derive the fixed point for the enqueue rate:

$$eq(t) = \eta\mu + \frac{N}{l} - \frac{\mu}{\delta}(x(t-\tau)-d_t)^+ \to \mu, \tag{18}$$

as $t \to \infty$.

Note while, we proved stability assuming that the feedback delay τ is a constant and the same value for all the senders, the proof works even if the senders have different time-varying feedback delays (see Corollary 3.2 in [49]). The modified stability criterion in this case is $\delta > \frac{2}{3}\cdot\tau^*$, where τ^* is the maximum feedback delay across all senders.

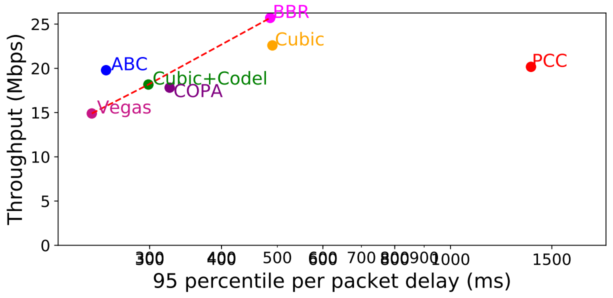


Figure 16: **Throughput and 95th percentile delay for a single user in WiFi —** We model changes in MCS index as bownian motion, with values changing every 2 seconds. We limit the MCS index values to be between 3 and 7. ABC outperforms all other schemes.

D Wi-Fi Evaluation

In this experiment we use the setup from Fig. 9a. To emulate movement of the receiver, we model changes in MCS index as brownian motion, with values changing every 2 seconds. Fig. 16 shows throughput and 95th percentile per packet delay for a number of schemes. Again, ABC outperforms all other schemes achieving better throughput and latency trade off.

E Low Delays and High Throughput

Fig. 17 shows the mean per packet delay achieved by various schemes in the experiment from Fig. 8. We observe the trend in mean delay is similar to that of 95th percentile delay (Fig. 8b). ABC achieves delays comparable to Cubic+Codel, Cubic+PIE and Copa. BBR, PCC Vivace-latency and Cubic incur 70-240% higher mean delay than ABC.

F ABC vs Explicit Control Schemes

In this section we compare ABC's performance with explicit congestion control schemes. We consider XCP, VCP, RCP and our modified implementation of XCP (XCP$_w$). For XCP

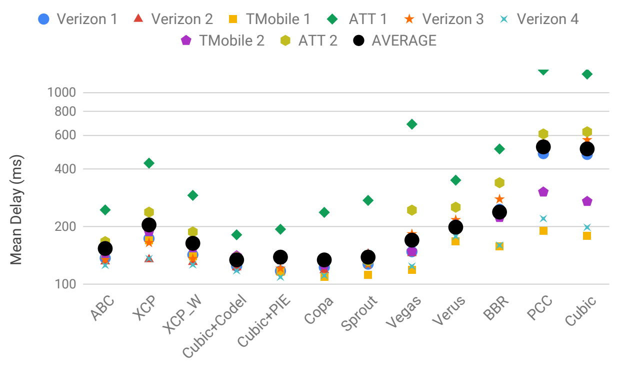
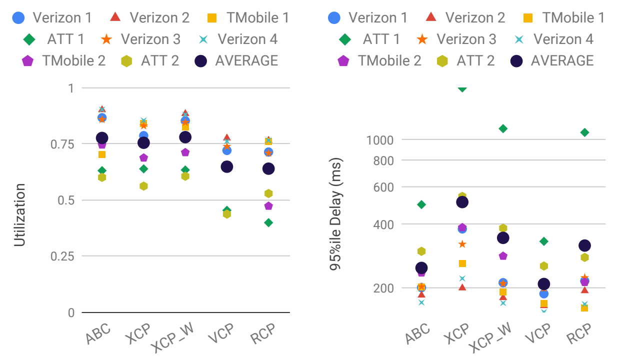


Figure 17: **Utilization and mean per-packet delay across 8 different cellular network traces —** On average, ABC achieves similar delays and 50% higher utilization than Copa and Cubic+Codel. BBR, PCC, and Cubic achieve slightly higher throughput than ABC, but incur 70-240% higher mean per-packet delays.



(a) Utilization (b) Delay

Figure 18: **ABC vs explicit flow control —** ABC achieves similar utilization and 95th percentile per-packet delay as XCP and XCP$_w$ across all traces. Compared to RCP and VCP, ABC achieves 20% more utilization.

and XCP$_w$, we used constant values of $\alpha = 0.55$ and $\beta = 0.4$, which the authors note are the highest permissible stable values that achieve the fastest possible link rate convergence. For RCP and VCP, we used the author-specified parameter values of $\alpha = 0.5$ and $\beta = 0.25$, and $\alpha = 1$, $\beta = 0.875$ and $\kappa = 0.25$, respectively. Fig. 18 shows utilizations and mean per packet delays achieved by each of these schemes over eight different cellular link traces. As shown, ABC is able to achieve similar throughput as the best performing explicit flow control scheme, XCP$_w$, without using multibit per-packet feedback. We note that XCP$_w$'s 95th percentile per-packet delays are 40% higher than ABC's. ABC is also able to outperform RCP and VCP. Specifically, ABC achieves 20% higher utilization than RCP. This improvement stems from the fact that RCP is a rate based protocol (not a window based protocol)—by signaling rates, RCP is slower to react to link rate fluctuations (Figure 19 illustrates this behavior). ABC also achieves 20% higher throughput than VCP, while incurring slightly higher delays. VCP also signals multiplicative-increase/multiplicative-decrease to the sender. But unlike ABC, the multiplicative increase/decrease constants are fixed. This coarse grained feedback limits VCP's performance on time varying links.

Fig. 19 shows performance of ABC, RCP and XCP$_w$ on a simple time varying link. The capacity alternated between 12 Mbit/sec and 24 Mbit/sec every 500 milliseconds. ABC and XCP$_w$ adapt quickly and accurately to the variations in bottleneck rate, achieving close to 100% utilization. RCP is a rate base protocol and is inherently slower in reacting to congestion. When the link capacity drops, RCP takes time to drain queues and over reduces its rates, leading to under-utilization.

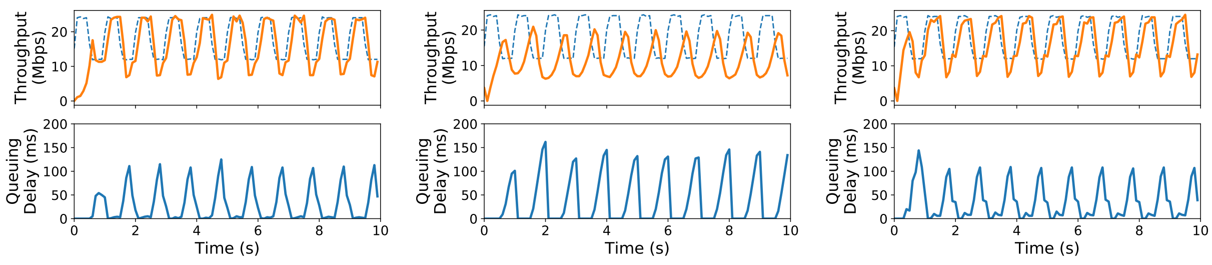
G Other experiments

Application limited flows

We created a single long-lived ABC flow that shared a cellular link with 200 application-limited ABC flows that send traffic at an aggregate of 1 Mbit/s. Fig. 20 shows that, despite the fact that the application-limited flows do not have traffic to properly respond to ABC's feedback, the ABC flows (in aggregate) are still able to achieve low queuing delays and high link utilization.

ABC's sensitivity to network latency:

Thus far, our emulation experiments have considered fixed minimum RTT values of 100 ms. To evaluate the impact that propagation delay has on ABC's performance, we used a modified version of the experimental setup from Fig. 8. In particular, we consider RTT values of 20 ms, 50 ms, 100 ms, and 200 ms. Fig. 21 shows that, across all propagation delays, ABC is still able to outperform all prior schemes, again achieving a more desirable throughput/latency trade off. ABC's benefits persist even though schemes like Cubic+Codel and Cubic+PIE actually improve with decreasing propagation delays. Performance with these schemes improves because bandwidth delay products decrease, making Cubic's additive increase more aggressive (improving link utilization).



(a) ABC (b) RCP (c) XCP$_w$

Figure 19: **Time series for explicit schemes —** We vary the link capacity every 500ms between two rates 12 Mbit/sec and 24 Mbit/sec.The dashed blue in the top graph represents bottleneck link capacity. ABC and XCP$_w$ adapt quickly and accurately to the variations in bottleneck rate, achieving close to 100% utilization. RCP is a rate base protocol and is inherently slower in reacting to congestion. When the link capacity drops, RCP takes time to drain queues and over reduces its rates, leading to under-utilization.

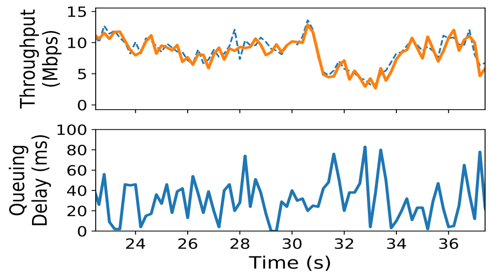
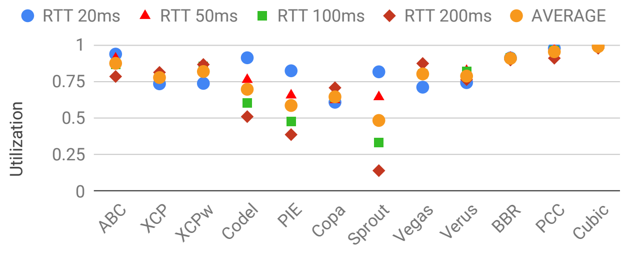
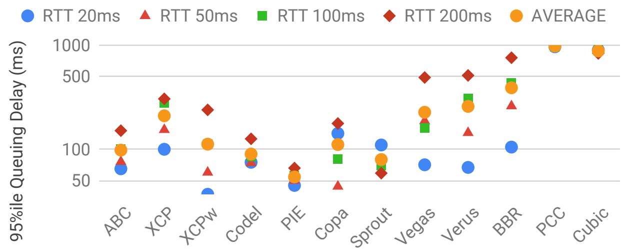


Figure 20: **ABC's robustness to flow size —** With a single backlogged ABC flow and multiple concurrent application-limited ABC flows, all flows achieve high utilization and low delays.



(a) Utilization



(b) per-packet queuing delay

Figure 21: **Impact of propagation delay on performance —** On a Verizon cellular network trace with different propagation delays, ABC achieves a better throughput/delay tradeoff than all other schemes.

AmphiLight: Direct Air-Water Communication with Laser Light

Charles J. Carver[†1], Zhao Tian[†1], Hongyong Zhang[2], Kofi M. Odame[2],
Alberto Quattrini Li[1], and Xia Zhou[1]
[1]Department of Computer Science, Dartmouth College, [2]Thayer School of Engineering, Dartmouth College
[†]Co-primary authors
{ccarver, tianzhao}@cs.dartmouth.edu,
{hongyong.zhang.ug, kofi.m.odame, alberto.quattrini.li, xia.zhou}@dartmouth.edu

Abstract

Air-water communication is fundamental for efficient underwater operations, such as environmental monitoring, surveying, or coordinating of heterogeneous aerial and underwater systems. Existing wireless techniques mostly focus on a single physical medium and fall short in achieving high-bandwidth bidirectional communication across the air-water interface. We propose a bidirectional, direct air-water wireless communication link based on laser light, capable of (1) adapting to water dynamics with ultrasonic sensing and (2) steering within a full 3D hemisphere using only a MEMS mirror and passive optical elements. In real-world experiments, our system achieves static throughputs up to 5.04 Mbps, zero-BER transmission ranges up to 6.1 m in strong ambient light conditions, and connection time improvements between 47.1% and 29.5% during wave dynamics.

1 Introduction

The underwater world is still largely unexplored, yet surveying and monitoring submerged sites is fundamental for many applications including archaeology [18], biology [40], and disaster response [42]. It is generally recognized that using multiple heterogeneous cyberphysical assets – e.g., flying vehicles for a bird's eye view and underwater sensors and vehicles for informed data collection – will advance such efforts [19, 28, 59]. One of the challenges for underwater autonomous deployments is limited communication between assets underwater and in the air. This hinders the situational awareness and coordination of underwater vehicles, data-processing, and human supervision [47]. One conventional strategy is to periodically let the underwater vehicle surface to share data [54], which is inefficient due to time not being spent on the task. Another strategy is to deploy an infrastructure (e.g., network of buoys) at the water surface, connected to both the underwater assets (via acoustic transducers, completely in the water) and the ground station (via tethering or WiFi [30]). This deployment configuration increases the cost and logistical overhead, limiting the overall scalability [39].

We seek solutions that support direct wireless communi-

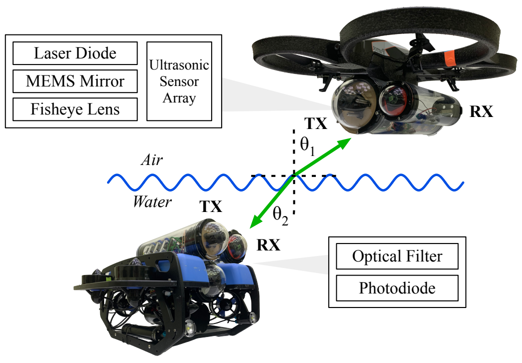


Figure 1: *Our envisioned application scenario of air-water communication allowing aerial drones and underwater robots to communicate directly and bidirectionally.*

cation between air and underwater nodes without the need of surface relays. Existing wireless communication technologies, however, mainly focus on a *single* physical medium and thus do not effectively cross the physical air-water boundary, impairing communication performance. As examples, acoustic communication is the mainstream for underwater scenarios but does not cross the air-water boundary since acoustic waves are mostly reflected by the air-water interface [49]; on the other hand, wireless technologies using radio frequencies (RF) are widely deployed in the air but not underwater since radio signals suffer from severe attenuation in the water (3.5–5 dB/m) and result in short communication ranges [45, 70]. A recent work [67] designs a direct water-air communication link by combining an acoustic link in the water and RF sensing in the air. Nevertheless, this method only enables a unidirectional link (from water to air), supports only centimeter-level distances above the water, and achieves severely low data rates (400 bps) that are insufficient for most underwater monitoring applications [26].

In this paper, we study the use of laser light to build a high-bandwidth, bidirectional air-water communication link (Fig. 1). Light is the most suitable medium because the majority (90%) of its energy penetrates the air-water interface with only less than 10% energy reflected back.[1] Compared to acoustics, light communication supports much shorter communication latency with faster propagation speeds. Compared to RF, it endures much lower attenuation in the water. In particular, light in the blue/green range (420 nm – 550 nm) attenuates less than 0.5 dB/m in water [5,45]. We specifically consider blue/green *laser* light because of its superior communication properties: (1) nanosecond-level switching speed, (2) narrow (5–10 nm) spectral power distribution,[2] allowing optical energy to be concentrated to the wavelength range associated with the smallest attenuation in the water/air, and (3) low beam divergence maximizing the energy efficiency and enhancing communication distance. Gbps-level data rates have already been demonstrated using laser light for air-water communication, albeit assuming calm water and with bulky benchtop equipment that are not portable to drones or robots [21,71].

The key contribution of our work is addressing numerous practical challenges currently unsolved (even with the assumption that the locations of the nodes – one underwater and one in air – are fixed and known) and providing a system framework, *AmphiLight*, for a robust laser-based air-water communication link. *First*, we judiciously design the basic communication link to overcome issues of existing laser hardware and improve its portability for communication. *Second*, to handle strong ambient light interference, we exploit the narrow spectral power distribution of laser light by placing a narrow optical filter in front of an ultra-sensitive receiver (silicon photomultiplier) to filter out ambient light and maintain sufficient signal-to-noise ratios (including at meter-level distances with low-power laser diodes). *Third*, to adapt to environmental dynamics, we propose a new optical system to enable precise, full-hemisphere laser steering using low-cost, portable hardware. It couples a fine-grained MEMS mirror with a miniature fisheye lens to achieve $\pm 90^\circ$ steering range in two dimensions. *Finally*, we address water dynamics by augmenting the link with ultrasonic sensing and a forecasting method. The ultrasonic sensor array at the transmitter samples the depth of a small number of locations at the air-water interface. These depth values are used to reconstruct a continuous water surface and compute the optimal incident point for the transmitter to steer the laser beam to reach the receiver.

We implement a proof-of-concept AmphiLight prototype using off-the-shelf hardware. Our prototype consists of the following elements: (1) a self-contained, waterproof laser transmitter utilizing a microcontroller, FPGA, MEMS mirror, and passive optical components; (2) an array of low-cost, ultrasonic depth sensors for reconstructing the water's surface; (3) a waterproof laser receiver capable of detecting the nanosecond laser pulses. We conduct experiments in various settings to examine both link performance and robustness. We summarize our key findings as below:

- AmphiLight achieves bidirectional, 5.04 Mbps throughputs with BERs less than 10^{-3} up to 6.5 m in the air and 2.5 m underwater;
- AmphiLight adapts to wave dynamics (10 – 12 cm wave amplitude and 1-Hz wave frequency) with a 47.1% throughput improvement over no laser steering;
- AmphiLight is robust against environmental factors including strong sunlight and air/water turbulence at meter ranges;
- The ultrasonic sensing achieves an accuracy of 1.5 cm in the air and 0.5–1.0 cm in the water.

2 System Challenges

Despite the potential of green-blue laser light for direct air-water communication, we face numerous systems challenges in achieving high link speed and link reliability.

Laser Hardware Limitations. Although laser diodes (LDs) are small and relatively inexpensive – making them strong contenders for mobile applications – integrating them into portable platforms for high-speed communication is challenging due to heating and power issues. Our experiments with off-the-shelf LDs show that their temperature rises over time when constantly on.[3] The temperature rise causes the central emission wavelength to shift by a few nanometers [7], which is undesirable as shown later in §3.1. Better heat dissipation requires dedicated temperature controllers and active heatsinks, which are bulky (9 lbs), expensive ($\geq$\$1000), and power hungry (up to 60 W) [8].

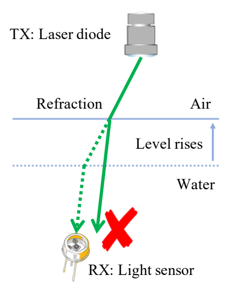
Additionally, commercial LDs are limited in terms of their optical output powers and wavelength availability. Specifically, blue and green TO-Can LDs are typically limited to 450 nm and 520 nm with optical powers between 30 mW and 140 mW and high power options between 900 mW and 3 W [9]. To maintain stable output power, LDs are typically powered with bench-top power supplies with current and voltage limits or mobile drivers that do not support fast modulation bandwidths (e.g., only up to 2 MHz [16]). The power consumption of low-power LDs ranges from a few milliwatts to multiple watts, making mobile-friendly micro-controllers incapable of consistent, safe, and efficient LD operation.

Ambient Light Interference. Given the sparse availability of blue/green LDs, low-power options are the only choice for mobile applications. Thus, strong ambient light, especially in outdoor scenarios, imposes a nontrivial challenge of maintaining high data rates with reasonable signal-to-noise ratios (SNR) at meter-level distances. Even worse, outdoor sunlight can easily saturate sensitive photodiodes (PDs) at the receiver, making it unresponsive to encoded light changes

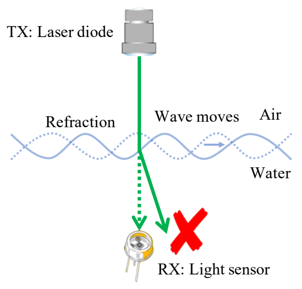
[1] If the incident angle is less than 50°.

[2] By contrast, the spectral power of light emitted from an LED can span up to 100 nm [2,68].

[3] When powering a PLT3520 LD with a Thorlabs S1PLM38 passive heatsink using 6 V and 150 mA, the LD casing temperature increased from 81 °F to over 145 °F, measured by a Lasergrip 774 infrared thermometer.



(a) Level change



(b) Wave

Figure 2: Water dynamics degrade the link reliability due to light refraction at the air-water interface. (a) Precipitation and tide can raise the water level, which permanently translates the refracted light and disrupts the aligned link. (b) Periodic waves can swing the refracted light, resulting in recurrent misalignment with the receiver.

from the transmitter. To illustrate this point, we measured the SNR of an off-the-shelf PD (OPT101) under varying ambient light intensities. Specifically, we collocate a LX1330B light meter with OPT101 and place a 140 mW LD and Osram 5500T03 LED 20 cm away, where the LED light emulates ambient light interference. Next, we vary the intensity of the LED and measure the resulting SNR at the PD. As the LED illuminance approaches values associated with outdoor ambient light (e.g., $\geq$10,000 lx in indirect sunlight), the SNR quickly drops below 3 dB (specifically, 3.2 dB at 5700 lx and 0.6 dB at 8070 lx). Furthermore, PDs capable of detecting low-level light need to have sufficiently high gain, making them susceptible to saturation under intense ambient light (the OPT101 became saturated when the LED intensity approached 14,500 lx).

Laser Beam Steering. Supporting arbitrary underwater/aerial robot locations demands precise steering of the narrow laser beam in a wide range. Existing laser steering mechanisms, however, face a fundamental tradeoff between steering range and granularity. Traditionally, FSO beam steering [37, 44, 58, 60, 64, 75] uses mechanical gimbals for 360° coarse-grained steering and then additional mechanisms for secondary, fine-grained adjustments [36, 46, 57]. Although mechanical gimbals can support a large angular steering range, they are bulky, imprecise, and not intended for use in mobile settings. On the other hand, the mechanisms used for fine-grained steering (e.g., microelectromechanical-systems (MEMS) mirrors [29, 41, 50, 51, 77], acousto-optic deflectors (AODs) [60, 69], tunable lenses [22, 52, 82, 82]) only achieve millirad/single degree steering ranges [46], constraining the receiver location to a narrow cone around the transmitter.

Environmental Dynamics. In real world environments, such as lakes or oceans, the water's surface is dynamic, rendering a laser link unsustainable due to refraction at the air-water interface. The impact of water dynamics is twofold: (1) A change in water level caused by precipitation or a tide can disrupt the optical link permanently. For example, a rise in the water level will move the incident point on the surface to a new position if the incident angle is not 0°. Consequently, the refracted light will be translated and miss the underwater receiver (Fig. 2(a)). Based on geometry, the horizontal displacement of the light beam is $\Delta h(\tan\alpha - \tan\beta)$, where Δh is the level change, α is the incident angle, and β is the angle of refraction. A level change of 1 m[4] with a 30° incident angle results in 17 cm displacement of the beam, far beyond the diameter of common light sensors (a few mm); (2) Periodic surface waves caused by wind or moving objects can swing the refracted light around the receiver. The oscillation causes the optical link to deviate from the receiver (Fig. 2(b)). Our experiment shows that waves with $\sim$10 cm peak-to-peak amplitudes make the link unavailable for $\sim$70% of the time.

3 Basic Laser Link Design

We present the basic laser communication link design able to (1) achieve sufficient data rates (i.e., Mbps for underwater drone communication and sensing) with off-the-shelf laser diodes and (2) support a hemispherical steering range to connect the transmitter and receiver at arbitrary locations.

3.1 Transmitter & Receiver

Transmitter. To support Mbps throughputs and low energy consumption – important tradeoff design for underwater drones – we adopt the DarkLight concept in [65]. Specifically, DarkLight applies overlapping pulse position modulation (OPPM), where data is encoded into the position of the rising edge of a light pulse within a symbol. We extend DarkLight to LDs, leveraging LD's fast switching speeds to increase the data rate while still maintaining a low duty cycle. This leads to a significant improvement in throughput from Kbps with LEDs to Mbps with LDs, as shown later in §6. Reducing the duty cycle removes the need for a dedicated temperature controller as the laser will remain off the majority of the time. Furthermore, a low duty cycle reduces the power consumption issues typically associated with laser communication, allowing us to power the LD with a microcontroller without sacrificing the data rate.

Even though OPPM is the most suitable choice for sustained communication[5] given the current laser hardware limitations, the AmphiLight framework is general and can be combined with other modulation schemes. As advances in LD hardware will better address heating[6] and power issues in the future, other modulations schemes such as OOK or OFDM can be easily integrated into AmphiLight to further boost link data rates. However, with higher-power modulation schemes, the effects of turbulence, especially over long distances, might degrade the overall link quality. Regard-

[4]For reference, the tidal range (height difference between high tide and low tide) can reach 16 m [11].

[5]If the communication between drones is intermittent, higher-power modulation schemes should not raise the temperature to dangerous levels, even with only passive cooling.

[6]In the case of an underwater drone, the surrounding water could be leveraged to passively cool the hardware without requiring additional power or expensive components.

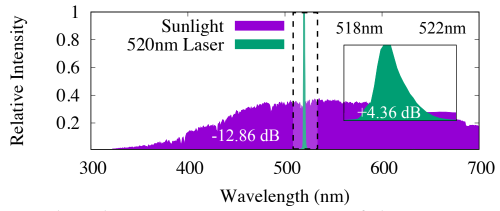


Figure 3: *The relative intensity spectrum of the sun compared to a low-power LD. Utilizing a narrow bandpass filter from 518 nm to 522 nm, the SNR can be increased from -12.86 dB to 4.36 dB.*

less, fully leveraging the GHz switching speeds of LDs, an OOK implementation could achieve throughputs in the Gbps range with only a few mJ/bit.

Receiver. We address the key challenge of the receiver design – to extract signals from low-power LDs amid strong ambient light interference while maintaining meter-level distances – via two design elements. First, we add a narrow optical bandpass filter ($30 – $200) that allows only the narrow wavelength range of the laser light (confined only to a few nm [53]) to pass, and filter out the majority of the ambient light energy and significantly boost the signal-to-noise ratios (SNRs). As an example, Fig. 3 plots the spectral power distribution of outdoor sunlight (measured on a sunny noon in August, 2019), as well as that of a low-power LD [13], measured by a Thorlabs CCS100 spectrometer. We observe that the weak laser light is buried in the strong sunlight. Nevertheless, adding an off-the-shelf bandpass filter [12] with a ± 2 nm bandwidth, we drastically improve the SNR. Additionally, spectral filtering also addresses the problem of sensor saturation under strong ambient light.

Second, we utilize an ultra-sensitive silicon photomultiplier (SiPM) light sensor, i.e., an array of avalanche photodiodes (APDs), with high gains, large active areas, and large angular responses [14].[7] Given the SiPM's significantly higher gain compared to traditional light sensors, we are able to maintain a sufficiently high SNR even with low-powered LDs at meter-level distances. We further increase the SNR by using an RF amplifier with a DC-bias cutoff, allowing us to amplify only the low-power laser light.

3.2 Full-Hemisphere Beam Steering

We adapt the fine-grained steering mechanism from FSO by expanding its limited steering range with a judiciously-designed optical circuit. Specifically, we combine a small-angle MEMS mirror with a miniature fisheye lens [24] to enlarge the small-angle steering to $\pm 90^\circ$ in two dimensions.

As shown in Figure 4(a), a fisheye lens is a combination of wide-angle lenses typically used to create hemispherical images for photographs. Fisheye lenses concentrate light rays coming from a full hemisphere to a small image plane at the focal length, limited by the form factor of digital camera image sensors. We exploit this optical feature to expand the

[7]We measured the SiPM's SNR to be between 13.03 dB and 13.95 dB between -70° and 80°.

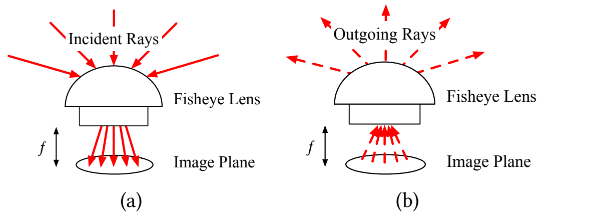


Figure 4: *(a) Light enters the fisheye lens and is projected onto a small image plane, compressing the wide incoming light directions into a smaller range. (b) We consider the inverse of the propagation path to enlarge a narrow steering range to full hemisphere.*

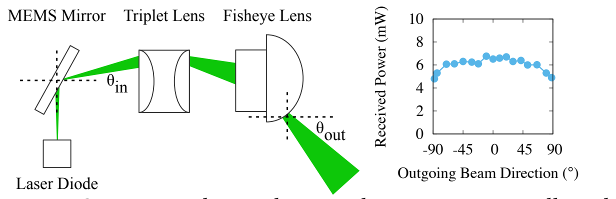


Figure 5: *Our proposed optical circuit design, using a small-angle MEMS mirror and fisheye lens. Not only can we achieve a full $\pm 90^\circ$ range in two dimensions, but the received power only deviates by 28% at extreme angles.*

narrow steering range of MEMS mirror. Specifically, given the path symmetry of light propagation, we consider the inverse direction of the light path by sending a light ray through the image plane. This leads to an outgoing light ray steered to a larger irradiance angle (Fig. 4(b)), thus expanding the small input steering range to an entire hemisphere.

Fig. 5 shows the optical circuit for laser beam steering. An achromatic triplet lens [83] is added to keep a constant focal point on the fisheye lens (i.e., correcting for spherical aberrations) [15]. It also concentrates the outgoing light ray from the MEMS mirror to the image plane of fisheye lens to match the desired inverse propagation path.

4 Addressing Water Surface Dynamics

Armed with the basic link design, we now set out to address challenges from dynamics at the air-water interface, aiming to improve link robustness in practical settings. To mitigate the misalignment caused by water dynamics, a straw-man approach is to expand/diffuse the laser beam to keep the receiver within the light coverage during water dynamics. This approach, however, greatly lowers the energy efficiency of communication and demands high-power LDs to support meter-level distances. Another approach is to blindly steer the laser beam and scan all directions to search for the direction that reaches the receiver. The resulting overhead to scan the whole steering range (up to hundreds of ms with existing MEMS mirrors), however, reduces the link throughput. It also requires a feedback channel from the receiver, which may be equally unavailable due to misalignment.

Instead, we consider a more proactive approach where the system continuously senses the condition (both the water level and the shape of the wavy surface) of the air-water interface, computes the optimal direction to reach the receiver,

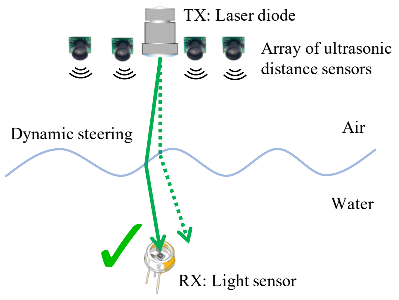


Figure 6: Addressing water dynamics by continuously sensing the water with an array of ultrasonic sensors, interpolating the surface, computing the optimal path to the receiver, and steering the laser.

and then steers the laser beam correspondingly to sustain the link's connection (Fig. 6).

4.1 Sensing Waves

To sense the water surface condition, we start by examining the efficacy of existing techniques. Vision-based methods with depth cameras have been widely used to reconstruct the 3D shape of objects. These methods, however, are unable to sense the shape of water surfaces because light mostly penetrates the air-water interface and reflects almost no light for the depth camera to reconstruct the surface. Our experiments with an Intel RealSense D435i depth camera shows that depth information is only correct when a piece of paper is placed on the water surface. Alternatively, one can consider RF-based methods, i.e., mmWave radar which has been shown to sense the distance from air to water at μm-level accuracy [67]. Given the severe attenuation of RF signals in the water, however, RF-based methods cannot be applied to underwater transmitters for sensing the water surface. Additionally, reconstructing the water surface requires an array of mmWave radars that can cost thousands of dollars.

The above exploration leads us to consider the acoustic medium. Specifically, we consider ultrasonic distance sensors to avoid interference from ambient noises. Ultrasonic distance sensors work in both air and water, and thus can be used by both aerial and underwater transmitters to sense the air-water interface and adapt the outgoing laser beam direction. Additionally, the accuracy of ultrasonic distance sensors are on the mm-level and are affordable (e.g., \$1 each).

Depth Sampling via Ultrasonic Sensing. To sense the shape of the water surface, a single ultrasonic sensor is insufficient. Instead, we employ an array of M sensors that are uniformly distributed on the transmitter plane. Because all sensors operate at the same acoustic frequency and are close to each other, simultaneous measurements cause interference. Therefore, we instruct the sensors to sample the distance sequentially. The sequential measurements result in a sensing latency that grows linearly with the number of sensors and proportionally to the distance between the transducers and water surface. In our implementation with 16 sensors, generating each snapshot of the surface is approximately 50 ms (20 Hz frame rate) which can impair the efficacy of beam steering for faster waves.[8]

To lower the latency of the sensor array, we propose to forecast the height samples of the water surface. Instead of waiting for the readings from all sensors to be ready, we can forecast the distances based on historical data. It is possible to forecast the height of the water because water surface waves are periodic. Specifically, we output distances from all sensor positions once a new reading is available from a sensor (e.g., at time t_{cur}). If the readings from other sensors are not ready at that time, we will use the forecasted distances. Because the water wave is periodic, we can use the Fourier transform for forecasting, i.e., estimate the frequency and phase of the waves despite the variable latency. Specifically, we buffer a window of the most recent N readings for each sensor x_k ($k \in [0, N-1]$),[9] compute the Discrete-time Fourier Transform (DFT) in the window, estimate the period of the major frequency component T, and forecast the reading x_N by linear interpolation at time $t_{x_N} - T$. If the timestamp of x_N is ahead of t_{cur}, we will linearly interpolate between x_{N-1} and x_N. Our measurements show that the forecast distances using historical readings align well with the measured distances. Forecasting reduces the sensing latency of each frame (i.e., time period between adjacent frames) to 1/16 of the non-forecast method, approximately 3 ms. Since the movement of the water waves between frames is the source of sensing errors, the forecasting error is 1/16 of the non-forecast error. This forecasting method assumes a single major frequency in the water waves. For more complicated waves, in the future we can investigate advanced forecasting methods, such as ARIMA and RNN (which require training and higher computational cost).

Reconstructing Wave Surface. To reconstruct a continuous wave surface for every frame, we need to interpolate between the discrete distance samples output by the array. We adopt a bicubic surface model [33] to fit the outputs:

$$h(x,y) = \sum_{i=0}^{3}\sum_{i=0}^{3} a_{ij} x^i y^j \,,$$

where (x,y) is the coordinate on the horizontal plane relative to the center of the sensor array, $h(x,y)$ is the height of wave at (x,y), and a_{ij}'s are the parameters of the surface. Bicubic surface is widely used in 2D interpolation. We choose this model for its shape flexibility and computational simplicity. We fit the model using linear regression, which is computationally inexpensive and suitable for real-time reconstruction. The linear system is $\boldsymbol{h} = \boldsymbol{X}\boldsymbol{\beta} + \beta_0$, where $\boldsymbol{h}$ is a vector of the measurement distances by each sensor, and $\boldsymbol{X}$ is a matrix that is constructed by the coordinates of the

[8]Typical frequency of water surface waves is 0.1 Hz–3 Hz [61].

[9]$N = 256$ in our implementation. Longer windows can produce finer-grained frequency resolution in the frequency domain and better forecast accuracy, yet entail longer buffers and higher computational cost.

sensors. Put formally, $\boldsymbol{X}$ is calculated as:

$$\boldsymbol{X} = \begin{bmatrix} x_1^3y_1^3 & x_1^3y_1^2 & x_1^3y_1 & x_1^3 & \cdots & y_1^3 & y_1^2 & y_1 \\ x_2^3y_2^3 & x_2^3y_2^2 & x_2^3y_2 & x_2^3 & \cdots & y_2^3 & y_2^2 & y_2 \\ & & & & \vdots & & & \\ x_M^3y_M^3 & x_M^3y_M^2 & x_M^3y_M & x_M^3 & \cdots & y_M^3 & y_M^2 & y_M \end{bmatrix},$$

where each row is for a sensor. The coefficients $\boldsymbol{\beta}$ and β_0 are the parameters of the surface, i.e., $\boldsymbol{\beta} = \{a_{ij} | i, j \in \{0,1,2,3\}\} - \{a_{00}\}$.[10] The model, therefore, contains 16 parameters[11] in total. With M ultrasonic sensors and thus M measurements, we employ regularized linear regression to prevent overfitting and the loss function is

$$f(\boldsymbol{\beta}, \beta_0) = \sum_{i=1}^{M} \left(h_i - \beta_0 - \sum_{j=1}^{15} \beta_j \boldsymbol{X}_{ij} \right) + \lambda \sum_{j=1}^{15} \beta_j^2,$$

where λ is a hyper-parameter that controls the penalty on the parameters. In our implementation, $\lambda = 5 \times 10^{-5}$.

4.2 Computing the Incident Point

Once the shape of the surface wave is estimated, we next seek an incident point on the surface such that the refracted light can reach the receiver. The incident light and refracted light must be subject to Snell's law. However, this equation is intractable because of the trigonometric functions, i.e, we have to solve the incident point position numerically. First, we model the problem as an optimization problem (Fig. 7). For every possible point on the surface, we are able to determine the direction of the refracted light ($\vec{r}$) according to Snell's law and the direction from the incident point to the receiver ($\vec{t}$). If the discrepancy between the two directions (θ) is zero, the previous equation is exactly solved. Therefore, we are looking for the incident point that minimizes the discrepancy θ, i.e., maximize $\cos\theta$:

$$\text{Maximize} \qquad \cos\theta = \frac{\vec{r} \cdot \vec{t}}{|\vec{r}||\vec{t}|} \tag{1a}$$

$$\text{subj. to:} \qquad \vec{r} = p\vec{m} + \vec{n} \tag{1b}$$

$$p = \frac{p|\vec{m}|}{|\vec{n}|} = \frac{\sin\beta}{\sin(\alpha - \beta)} \tag{1c}$$

$$\vec{n} = \left(\frac{\partial h}{\partial x}, \frac{\partial h}{\partial y}, -1 \right) \tag{1d}$$

Here $\vec{n}$ is the surface normal unit vector, $\vec{m}$ is the direction vector of the incident light, n_{water} is the refractive index of water, and $\frac{\sin\alpha}{\sin\beta} = n_{\text{water}}$. To calculate $\vec{r}$, we first determine the direction of the surface normal ($\vec{n}$) according to 1d. Suppose the direction vector of the incident light is $\vec{m}$, which

[10] β_0 is another name for a_{00}.

[11] Although we estimate the model parameters without any prior knowledge, we could utilize the physics of dispersion [4] to model the constraints between the different parameters (e.g., between wavelength and wave speed). This could improve the accuracy of the model, but is beyond the scope of this paper.

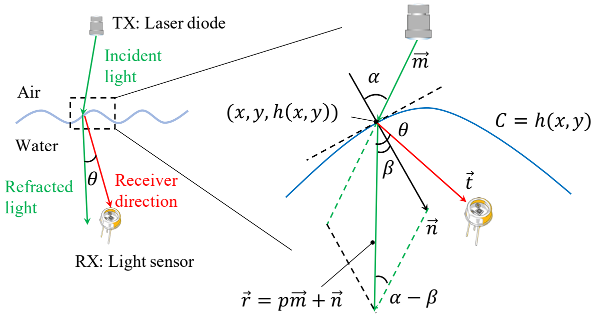


Figure 7: Geometric model of finding the optimal path to reach the receiver. We model finding the incident point on the surface such that the laser can reach the receiver as an optimization problem. This figure shows a single solution in the solution space, where the optimization happens over all possible solutions. We minimize the angle discrepancy (θ) between the refracted light ($\vec{r}$) and the target path that reaches the receiver ($\vec{t}$) subject to Snell's law which governs the relation between the angles of incidence (α) and refraction (β).

is also a unit vector. According to Snell's law, the incident light, the refracted light, and the surface normal are coplanar. The refracted light's direction can thus be written in the form of 1b, where p is calculated according to law of sines and Snell's law (1c). Assuming the transmitter is in the air and the receiver is underwater, n_{water} is the refractive index of water (we can take the reciprocal if the transmitter and receiver exchange positions). Notice that all the quantities are functions of (x, y).

Algorithm 1: Find outgoing beam direction

```
Input: $\boldsymbol{p}_0$: initial incident point, $\boldsymbol{r}$: receiver position, $\boldsymbol{b}$: surface
       shape parameters                                  // TX: (0,0,0)
Output: $(\gamma_x, \gamma_y)$: outgoing beam angle along $x$-axis and $y$-axis
$\alpha \leftarrow 0.01$                                 // learning rate
$\boldsymbol{p}_n \leftarrow \boldsymbol{p}_0$           // next point to test
$\varepsilon \leftarrow 0.005$                           // accuracy tolerance
$N \leftarrow 100$                                       // max iteration
for $i \leftarrow 1$ to $N$ do
    $\boldsymbol{p} \leftarrow \boldsymbol{p}_n$
    /* compute gradient and error                        */
    $\nabla, e \leftarrow$ gradient$(\boldsymbol{p}, \boldsymbol{r}, \boldsymbol{b})$
    if $e < \varepsilon$ then
        break
    $\boldsymbol{p}_n = \boldsymbol{p} + \alpha\nabla$
end
$z = h(\boldsymbol{p})$                                  // z coordinate on surface
$\gamma_x = \arctan(\boldsymbol{p}_x / abs(z))$
$\gamma_y = \arctan(\boldsymbol{p}_y / abs(z))$
return $(\gamma_x, \gamma_y)$
```

We solve the optimization problem by gradient ascent (Algo. 1). We compute the gradient ($\nabla\cos\theta$) following the chain rule. For each frame, we set the initial incident point as the incident point of the previous frame. The spatial and temporal continuity of the water wave and the observation of the optimization result show that the new incident point should be close to the previous one because the change of the surface shape is small between adjacent frames. All the computation, including the forecasting, surface reconstruction, and path finding, can be completed within 1 ms. Note that wave sensing, path finding, and beam steering occur in parallel with the data transmission, incurring no overhead on the optical link. Notably, our algorithm has the potential to fail if some of the characteristics are beyond the sensing capabilities of the ultrasonic array: wavelength is smaller than the separation between the ultrasonic sensors; wave frequency is higher than the sensing frequency. We discuss improvements to the sensing and reconstruction in §7.

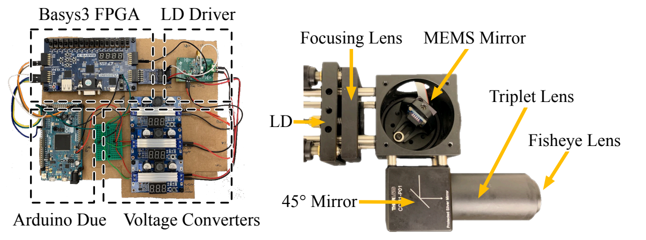
5 Prototype Implementation

Our prototype includes a transmitter, which encompasses the optical circuit, electronic circuit, modulation scheme, and ultrasonic sensor array, and a receiver that includes optical filtering and receiver hardware.

Transmitter. Our transmitter utilizes a small, mobile optical circuit relying on a single MEMS mirror and passive optical components to achieve a hemispherical steering range. The complete package is shown in Fig. 8(b). We mount a 140 mW TO-Can PLT3520D LD ($89) within a Thorlabs S1LM38 passive heatsink and focus the light to a 2° half-angle beam with an A110TM-A aspheric lens ($90). We use a 3.6 mm MEMS mirror mounted on an A7B1.1 actuator in a TINY20.4 package from MirrorcleTech ($798), providing roughly $\pm 6.6^\circ$ mechanical deflection on the x/y axes and 0.003° angular resolution.[12] The mirror is fixed to a MirrorcleTech mount and screwed into a Thorlab B5C1 optics mount. The cage cube platform is rotated 45° relative to the mounted LD. We mount a Thorlabs TRH254-040-A triplet lens ($78, $\approx$ 1% loss) and Sunnex DSL419B-NIR-F2.0 miniature fisheye lens ($99, $\approx$ 10.5% loss) within a Thorlabs SM1L20 lens tube. We decreased the distance between optimal components to maximize the final output power. Since the MEMS mirror's center point changes with gravity, we mount the lens tube on a Thorlabs CCM1-P01 45° mirror cube, allowing us to evaluate the parallel and perpendicular hemispheres without changing the mirror's alignment.

Fig. 8(a) shows the designed electronic circuit to transmit data. An Arduino Due ($32) processes the payload and split it into M-bit chunks. We implement an FPGA pulse timer on a Basys3 ($149) with a clock speed of 100 MHz. We establish a serial connection between the Arduino and the FPGA and transmit the required OPPM parameters (e.g., symbol length, slot width, pulse width) and processed payload to the FPGA. The FPGA parses the data and outputs the pulses on a single GPIO pin. To supply enough power to the LD, we utilize a TI LMG1020EVM-006 laser diode driver ($154) which also shortens the input pulse width up to 45%. To communicate between the FPGA and driver, we shift the logic levels with a Digilent 410-320 PMOD logic level shifter ($14). Finally, to

[12]Our MEMS mirror has a mechanical range $\pm 6.6090^\circ$ on the x-axis and $\pm 6.5586^\circ$ on the y-axis. Furthermore, the mirror's ADC has a resolution of 12 bits, making the angular resolution 0.003° on the x and y axes.



(a) Electronic Circuit (b) Optical Circuit

Figure 8: *(a) Electronic circuit, 20 cm × 10 cm × 7.5 cm and weighing 0.7 lbs, used to transmit nanosecond laser pulses. (b) Optical circuit to achieve full-hemisphere beam steering. The optical circuit is 11 cm × 11 cm × 5 cm and weighs 1.8 lbs.*

maintain mobility, we power all the components with the Arduino preprocessor and three voltage step-up converters.

To achieve a sufficient throughput in our prototype, we encode five bits per symbol and decrease the slot width to its minimum value (i.e., one clock cycle of 10 ns). To ensure the LD reaches its peak power within one pulse, we set the pulse width to 150 ns resulting in a duty cycle of 13.70%. Since SiPM's experience exponential decay, we add a 300 ns guard interval, T_G, at the end of each symbol for the sensor to reset. A 2 μs fixed delay occurs after the last symbol, giving the receiver time to differentiate between adjacent packets. Our modulation parameters enable a maximum throughput of 5.04 Mbps. Note that faster FPGAs exist (e.g., up to 600 Mhz clock speeds [17]) which would support shorter slot widths and consequently faster throughputs (up to 30.22 Mbps with a slot width of 1.7 ns).

Finally, as shown in Fig. 9, we implement our ultrasonic sensing array using 4×4 HC-SR04 sensors [6] ($4 each), each sensor 6 cm apart from adjacent nodes.[13] The 16 sensors measure the distance from the transmitter plane to the water surface sequentially. Currently, the sensing latency of every data frame is approximately 3 ms with forecasting at a distance of 20 cm above the water. Notably, as the distance increases, the measurement will become less accurate since the ultrasonic beam will cover a larger area and the sensing latency will be higher due to the increased propagation delay. In our implementation, we set the exit condition of our gradient ascent algorithm (Algo. 1) using an error tolerance of 0.005 radians. We implement the wave shape fitting and optical path determination using C++. We leverage the MLpack library for the regularized linear regression and FFTW for the forecasting. Finally, our ultrasonic sensor array can be replicated for underwater transmitter by employing cheap waterproof ultrasonic sensors [10] and we discuss various ways to reduce the sensing latency increase the wave sensing accuracy in §7.

Receiver. Given the effects of ambient light on the receiver SNR, we implement wavelength filtering tuned to the emission wavelength of our LD. Although bandpass filters are the

[13]This distance was determined given the 15° FoV of each sensor.

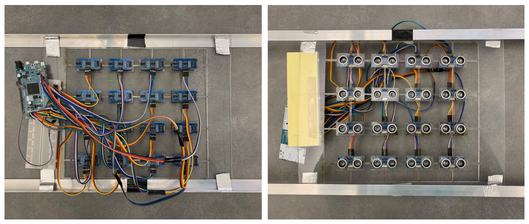

(a) Top of array (b) Bottom of array

Figure 9: *Top and bottom views of our ultrasonic array. The array is 26 cm × 36 cm × 5 cm and weighs 1 lbs.*

optimal choice given their narrow pass regions (e.g., 5 nm–10 nm), they require incident light to be nearly perpendicular to the filter for it to pass [1]. Since we aim to support arbitrary positions and orientations of the receiver, we instead complement our SiPM sensor's large angular response with a large-FOV colored glass filter – a Thorlabs FGV9 filter ($40) that passes light between 485nm and 565nm. In stronger ambient light conditions, a bandpass filter can be implemented with a concentrating lens (e.g., fisheye lens) in front of it (essentially using the same optical setup as the transmitter).

Finally, to detect the laser pulses, we use a KETEK PM3315-WB-B0 SiPM ($72, active area of 5 mm × 5 mm and 3 dB acceptance angle of 180°) connected to a KETEK PEPCB-EVAL bias board ($65). The SiPM is biased with 5.0 V and 5 mA, shortening the decay time of the sensor to around 60 ns. The signal from the bias board is fed to a MiniCircuits ZX60-P103LN+ RF amplifier ($70) powered with 5 V and 100 mA. We then utilize a Keysight MSOS254A 2.5 Ghz, 20 GSa/s oscilloscope to record the pulses received by the SiPM and demodulate the signals with MATLAB. We discuss our initial results at bringing real time demodulation to our transmitter via an analog circuit in §7.

6 Evaluation

We extensively evaluate the link performance and reliability of our air-water laser link in various settings.

6.1 Overall Performance

We first examine link throughput, bit error rates (BER), and communication distance. In these experiments, the transmitter and receiver are manually aligned. The angle of irradiance from the transmitter and the incident angle into the receiver are all zero unless otherwise specified. Experiments are mostly conducted in two settings: 1) a swimming pool (22.8 m × 10 m × 1.8 m); and 2) a water tank with clear fresh water (1.6 m × 1.75 m × 0.63 m). The ambient light intensity was between 380 lx and 450 lx throughout the experiments and the power consumption for each component can be found in Tab. 1.

Table 1: *Power consumption of various components.*

Component	Voltage (V)	Current (mA)	Power (mW)
TX (5.04 Mbps)	7.7	300	2300
Sensing Array	9.3	67	623
SiPM Bias	5.0	5	25

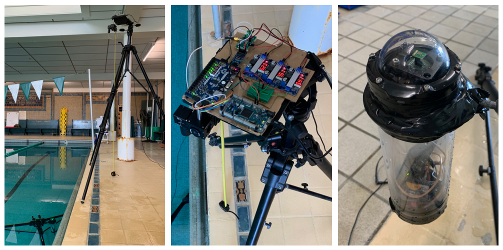

(a) Setup (b) TX (c) RX

Figure 10: *Experimental setup in a swimming pool.*

Evaluation Methodology. To capture nanosecond laser pulses, we use a Keysight MSOS254A oscilloscope to record the data, which is then transferred to a laptop for demodulation. This methodology, however, is limited by oscilloscope's buffer size. Specifically, the timing resolution of the oscilloscope is inversely proportional to the capture window size, meaning longer capture windows have inaccurate timing resolutions (i.e., laser pulses are missed or misaligned, causing demodulation errors and higher BERs).

To overcome this issue, we transition from our stationary oscilloscope receiver to an Arduino Due receiver. Specifically, we first obtain a mapping between the SNR and throughput/BER using the oscilloscope receiver to decode data in real time. Then when switching to the Arduino receiver, we instruct the transmitter to send a continuous wave for the Arduino to measure the SNR and map it to the corresponding throughput/BER. We have validated the accuracy of this methodology in experiments with a water tank in a lab setting, where we observe negligible differences between the estimated and actual throughput/BER with these two receivers. This methodology greatly facilitates our experiments during dynamic waves. Our ongoing work is to fabricate an analog circuit for detecting nanosecond light pulses without the need of an oscilloscope or expensive GHz-level ADC (§7).

Calm Water. We start by examining the link performance link under calm water. As shown in Fig. 10, we first place the receiver in a waterproof container with an SMA cable protruding from the bottom of the enclosure and connected to our oscilloscope on land. Second, we place the receiver underwater in a swimming pool and vary the distance to the surface from 0.5 m to 1.1 m. Third, we fix the transmitter to a tripod and place it on the bank of the pool, varying the height from 1 m to 2 m. Finally, we ensure the angle of irradiance out of the transmitter and incident angle into the receiver is below 10° at all times.

As shown in Fig. 11(a) and 11(b), we measure the average throughputs and BERs for each distance configuration. Throughout our experiments, the mean throughput was constantly above 5.03 Mbps and the BER below 0.01. Notably, the limited depth of the swimming pool and low ceiling height reduced the measurable range considerably. To further evaluate the potential of our link, we measure the range of the link separately in the air and in the water. To measure the

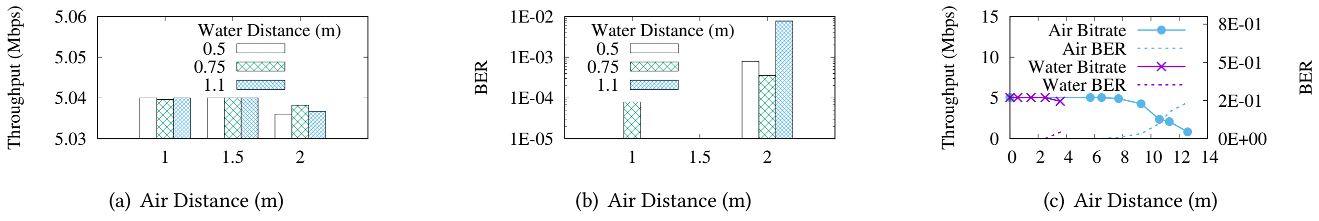


Figure 11: *Link performance under calm water. (a) and (b) plot the throughput and BER under various combinations of air and water distances (limited by the swimming pool setting). (c) plots the performance in pure air and water.*

range in the water, we place two mirrors at the long ends of a 50 cm fish tank and bounce the laser between them to increase the propagation path. As depicted in Fig. 11(c), we are able to achieve a zero-BER range in the air up to 6.5 m and a zero-BER range in the water up to 2.5 m. Consequently, we expect the air portion of our joint results to approach 6 m before the link degrades. Given the power loss associated with the light hitting the mirrors, however, we expect the underwater range to extend beyond our measured 2.5 m.

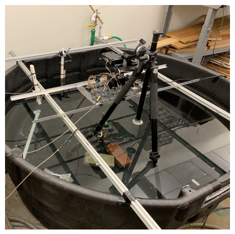
Next, to validate link bidirectionality, we place TX in a waterproof enclosure (weighs 4.8 lbs and measures 11.5 cm $\times$ 36 cm $\times$ 11.5 cm). To fit TX in the waterproof tube, we perform the following modifications: (1) remove the MEMS mirror and fix the laser directly to the triplet lens/fisheye lens tube; (2) replace the 140 mW PLT3520D LD with an 80 mW PLT5520B LD to mount within an adjustable focus enclosure; (3) power TX with a 9-V battery and decrease the LD driver supply to 6.5 V. We then fix the underwater distance to 17 cm and vary the air distance from 32 cm to 71 cm. We record the receiver's SNR and throughput in each direction. As shown in Tab. 2, the throughputs remained stable with a comparable decrease at 71 cm[14] corresponding to a mean BER of 0.00016 (RX underwater) and 0.00028 (TX underwater). Furthermore, the SNRs at each distance configuration varied, at most, by 2.6% indicating a symmetric link. The small discrepancy is most likely due to small offsets in beam alignment.

Table 2: *The difference in throughput and SNR for two directions.*

Air dist. (m)	Throughput % difference	SNR % difference
0.32	0.000%	2.6421%
0.54	0.000%	1.0572%
0.71	0.01191%	2.4117%

Dynamic Water. We now move on to examining link performance under water dynamics. We augment the TX with the ultrasonic sensor array and conduct the experiments in the water tank setting (Fig. 12) because of the ease of mounting the sensor array. We place the sensor array and TX 33 cm above the water surface in the middle of the tank to better simulate real-world conditions, e.g., the middle of a lake. To generate waves, we stir the water by hand for ten seconds, creating roughly uniform waves with amplitudes between 10 – 12 cm and wave frequency of approximately

[14]This range is smaller than our measured zero-BER range given the optical circuit modifications (i.e., lower power LD) and decreased transmitter power supply.



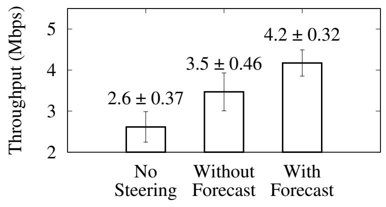


Figure 12: *Experiment setup and results with dynamic water surface.*

1 Hz. We then wait five seconds before recording the SNR on our Arduino for an additional ten seconds. To determine the connection percentage, we look at the percentage of time SNR is above 13.18 dB, the required threshold to maintain 5.04 Mbps throughput with 8×10^{-5} BER. To compute the throughput, we multiply the measured connection time by the corresponding mapped throughput.

We compare our method to two baselines: (1) *No steering*, where the direction of the light is fixed without any response to the waves, resulting in a frequent loss of link connection during wave dynamics; (2) *Wave sensing w/o forecasting*, which is a variant of our proposed sensing method in §4 without the forecasting, i.e., the system collects data from all ultrasonic sensors before estimating the wave surface and steering the laser beam. We test each method in ten trials. Fig. 12 compares the throughput under various methods, where error bars covering standard deviations are also included. We make the following observations: *First*, methods with wave sensing improves link throughput, achieving 29.5% and 47.1% increases compared to no steering. The improvement is due to the higher percentage of link connection with wave sensing. Without steering, the link is disconnected 48.13% of the time because of the periodic wave surface changes, whereas active sensing and laser steering improves the connection percentage to 82.80%. *Second*, between the sensing methods with and without forecasting, forecasting achieves an additional gain because of its reduction in sensing delay, resulting in faster adaptation of the beam direction to wave movement. *Third*, compared to prior work [67] which only supports throughputs up to 400 bps, our system maintains a throughput of 4.2 Mbps with OPPM during wave dynamics – a 10,500 times improvement.

6.2 Link Reliability

Types of Water Waves. While our prior experiments with dynamic water show the high potential of our sens-

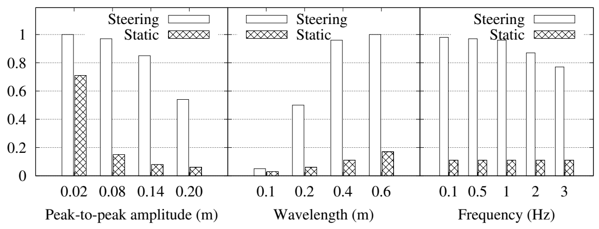


Figure 13: *Influence of water wave parameters on the reliability of the laser link.*

ing method, the results are under a single type of wave. It is practically difficult to precisely generate waves with known parameters and compare methods under exactly the same water waves. To gain a deeper understanding on the impact of different wave characteristics and compare methods more fairly, we build a simulator to generate synthetic waves and emulate the performance using various methods. Specifically, we simulate the water surface with a sinusoidal wave $h(x,y,t) = A\sin(\omega x + \varphi t)$[15], which is widely used in computer graphics to synthesize water waves [32,56]. To exclude the influence of modulation, we use the percentage of the throughput relative to the static link to represent the link reliability. In our simulation, TX and ultrasonic sensor array are placed 20 cm above the water surface. Because the half-angle of our laser beam is 2°, we consider the data to be decodable if the angle deviation between the RX and the laser beam center is less than 2°. We simulate the process for a whole period of the wave because the wave is periodic. By default, the wave's wavelength is 40 cm, peak-to-peak amplitude is 10 cm, and frequency is 1 Hz.

Fig. 13 compares our method to that without steering as we vary the wave characteristics including wave amplitude, wavelength, and frequency. We observe that our method significantly improves link reliability in most scenarios. We also gain two additional insights: (1) The performance of wave sensing degrades with increasing amplitude. This is due to the regularization in the surface interpolation, which gives a higher penalty for more curly waves. The performance also degrades with decreasing wavelength, which is constrained by the Nyquist sampling theorem. The distance between the acoustic sensor dictates the smallest wavelength that our method can reconstruct with high fidelity. A straightforward solution to address both degradations is to increase the density of the acoustic sensors at the expense of raising costs and complexity of sensor placement. A more sophisticated solution is to employ compressive sensing with incoherent sampling. We will discuss this more in §7. (2) The performance degrades gradually when the frequency increases in the normal range (0.1–3 Hz [61]). Even with the fastest waves (3 Hz), our method can sustain more than 75% of the throughput. If higher reliability is desirable, we can further lower the sensing latency by using frequency-division multiplexing among the acoustic sensors, which will increase

[15] $2A$ is peak-to-peak amplitude. $2\pi/\omega$ is wavelength. $2\pi/\varphi$ is frequency.

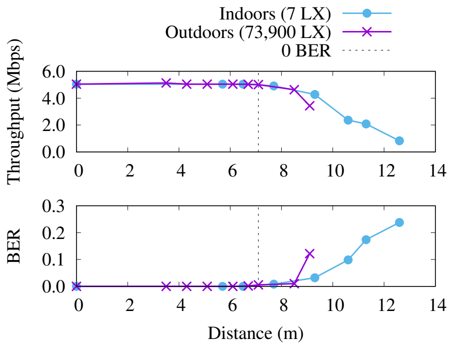


Figure 14: *Comparing link performance under two extreme light conditions: low light indoors and strong sunlight outdoors.*

the hardware complexity of the sensors. Compared to [67], which could support a sustained link up to 16 cm waves and no communication past 22 cm, our system maintains a ≥ 80% reliability up to 14 cm and 50% reliability up to 20 cm.

Sunlight. Given that our ultimate application scenario is outdoors, we evaluate the impact of strong sunlight on our link. Focusing on the air portion of the link, we vary the distance between the TX and RX in both indoor and outdoor scenarios and compare the resulting throughputs/BERs. We compare two extremes: a low-light condition indoors with illuminance between 5 and 7 lx and the strong-light condition (73.900 lx) outdoors, typical for direct sunlight at noon [3]. As shown in Fig. 14, the link performance under strong sunlight is similar to indoors within an 8 m link distance. It achieves zero-BER at 6.1 m distance outdoors, compared to zero-BER at 6.5 m indoors with low ambient light. The link distance under low light is slightly larger because of its higher SNR. This demonstrates that our RX design is robust to strong sunlight, benefiting from the narrow emission bandwidth of laser light and spectral filtering.

Air Turbulence. Air turbulence is known to affect light propagation due to the change in pressure/temperature [34, 73]. We next investigate its impact on the laser link and ultrasonic sensing. Given that pressure/temperature differentials are difficult to generate without dedicated equipment, we generate air current with a typical tabletop fan. Specifically, we place a laser and target 13 m apart. Next, we align the laser so it is in the middle of the target. We then place a fan on one side of the laser and turn it to its highest setting. Within one minute, we observed no changes in beam's alignment. The reason is that the distances supported by the system are immune to their effects, since air turbulence only degrades the signal quality at distances greater than 1 km [81].

Furthermore, we investigate the accuracy of our ultrasonic sensing with the inclusion of wind. We place a fan close to the acoustic path of the ultrasonic sensor that is continuously measuring the distance to the water surface. Results show that the measurement accuracy is not affected by the air flow or the acoustic noise caused by the wind. This is because the velocity of sound (340 m/s) is higher than wind (a few m/s) and the acoustic frequency of ultrasonic sensors is beyond that of wind in the audible range.

Water Turbulence. We also investigate whether water turbulence has an impact on the laser beam's propagation. We place a laser on one end of a 50-cm fish tank and a crosshair target on the other, aligning the two when the water is still. We then stir the water turbulently and observe the beam's propagation through the water. As long as the beam does not strike any object or the water's surface, we observed that the path remains throughout the turbulent flow. This result is expected given the short range tested. Similar to air turbulence, water turbulence is caused by changes in pressure and temperature [72, 78]. In other words, turbulent changes in temperature/pressure cause the refractive indices of eddies to change and bend light as it travels along its path. Water turbulence, however, can be much higher than in the atmosphere given the higher density levels underwater [35].

6.3 Ultrasonic Sensing

Our final set of experiments examines the accuracy of the ultrasonic sensing component.

Sensing in the Air. First, we evaluate the efficacy of ultrasonic sensing with a benchmark experiment. Because accurate reconstruction of liquid surfaces remains an open question in computer vision [20, 38, 56, 76] and also in ocean engineering where numerical models used for predicting ocean waves are at kilometer scale [23, 48, 66], we are unable to obtain the ground truth of an actual water surface. Therefore, we emulate the shape of waves by curving a piece of glossy poster paper by a peak-to-peak amplitude of 10 cm and wavelength of 24 cm. The sensor array is placed 20 cm above the emulated wave. We move the sensor array above different sections (e.g., crest and trough) of the wave to show the flexibility of the bicubic surface model. We manually measure the wave height at multiple positions as ground truth. Results show our sensing method and surface model successfully sense the height of the wave and reconstruct the wave shape with a median error of 1.5 cm.

Sensing in the Water. We validate the efficacy of underwater ultrasonic sensing with another benchmark experiment. We place a BlueRobotics Echosounder on a BlueRobotics BlueROV 2 underwater robot and record the acoustic distance over time. We sent 16 pings/second, fixed the gain to -4.4, and decreased the speed of sound from 1450 m/s to 1400 m/s to compensate for the chlorine water in the pool. Additionally, we use a tape measure to measure the distance from the sensor to the surface. After varying the distance between 56 cm and 89 cm, we observed an accuracy of 0.5 cm to 1 cm depending on the depth (the resolution decreased with larger ranges), which is similar to that in the air. Thus, we expect an array of sensors should perform similarly.

7 Discussion

Mobility. Currently, when our receiver changes its location (e.g., moves from the air to under water), the transmitter is unable to track and steer. However, since our transmitter is capable of steering to any arbitrary location within full hemisphere, we expect it can solve the mobility problem associated with air-to-water communication. The small beam divergence associated with laser light can enable the transmitter's laser beam to be reused for passive sensing. In that scenario, accurate retroreflectors can be collocated with each receiver and allow the transmitter to sense whether it has hit its target. The transmitter can then switch to transmission mode for a fixed duration and then resume sensing once complete. We can also combine the acoustic sensors and the retroreflected laser light to locate the receiver underwater to provide the coordinates of the receiver as input to determine the path while the receiver is moving.

Analog Receiver. Another ongoing work is to enable real-time demodulation with an analog circuit and microcontroller. Our analog circuit is a cascade of four stages: (1) amplifier, (2) envelope detector, (3) comparator and (4) demodulator. After amplification, the signal's long-term average amplitude is extracted with the envelope detector. Whenever the signal exceeds half of its long-term average amplitude, the comparator trips, and an OPPM pulse is registered. The threshold and original unbiased signal are then fed to a nanosecond-level comparator to determine whether the received voltage is an OPPM pulse. To determine the timing between pulses, we feed the output of the comparator to a time-to-digital converter (e.g., TI TDC7201-ZAX-EVM). After synchronizing the clocks of the transmitter and the receiver, we can use an off-the-shelf microcontroller (e.g., Arduino) to start the timer before each symbol. The comparator output is then wired to stop the time-to-digital converter, which stores timing information until encountering the next stop signal. To ensure that no pulses are lost, the microcontroller reads the previous timing information before sending the next start signal.

Improving Wave Sensing. Since the sensing accuracy affects the accuracy of the wave reconstruction, it also contributes to the accuracy of the laser beam steering. Therefore, if we can further improve the sensing accuracy and latency, then we can reduce the ultrasonic beam angle, utilize a lower power transceiver, and still maintain the reliability and throughput of the link. *First*, we can greatly increase the frame rate of the waving sensing by frequency division multiplexing. The current acoustic sensor array relies on sequential distance measurements to prevent interference. Assigning near sensors to different acoustic frequencies, however, can eliminate the interference at the cost of more expensive sensors with broader modulation bandwidth [43]. *Second*, given the form factor of ultrasonic sensors ($\sim$1 cm), we are unable to achieve a spatial resolution as high as that of cameras because of the limited number of depth samples. To overcome this limit, we can employ compressed sensing to reconstruct wave shape with finer granularity by incoherent sampling conditioning on the sparsity of the spatial frequency of water waves. The challenge of applying compressed sensing is its high computational cost [27] due to the real-time requirement of laser beam steering. *Third*, we can exploit the characteristics of the water waves in the time domain to improve the accuracy. For example, we can leverage Guassian process to model the similarity of frequencies at near positions of the wave.

Real-World Evaluation. We recorded various preliminary ocean dynamics to validate our AmphiLight methodology and simulation results. Specifically, we built a waterproof buoy to measure the frequency and peak-to-peak amplitude of waves in Barbados over a two-day period. From our results, we observed waves with an average frequency between 1 Hz and 1.5 Hz and peak-to-peak amplitudes between 5 cm and 8 cm. Furthermore, we measured the throughput of our system under the same wave conditions, and observed an average throughput of 3.1 Mbps corresponding to a connection time of only 61%. Comparing these measurements to our simulation results in Fig. 13, our AmphiLight system is capable of compensating for waves with these characteristics.

In terms of link range, the maximum distance achievable in a real-world environment depends on the laser and ultrasonic sensors. As for the laser, the limiting factor is the optical output power (i.e., the higher power the laser, the farther it will travel before attenuating). In our implementation, there was some optical output power loss in the optical circuit. Specifically, our fisheye lens had an optical loss of $\approx$10.5% and extensive scattering right after exiting the lens. Additionally, the sensing range of the ultrasonic transceivers was another limiting factor. In our current prototype, low-cost transceivers emit low-power waves with large beam divergences, limiting the link range to a few meters. To accommodate real-world scenarios, where the waves are dynamic and unpredictable, the laser power and sensing power should be increased and beam divergence decreased (e.g. < 1 degree) to maintain a sufficient range.

8 Related Work

Air-Water Communication. Air-water communication is a topic far less explored. An early work [55] in 1992 reported a laser link between an aircraft and a submerged submarine, though minimal details about the system design and implementation were disclosed. In the concept figure, the aircraft broadcasts a laser beam with large field of view, which potentially imposes a high energy budget. A later work [21] demonstrated a static laser link achieving 5.3 Gbps data rate over a 5-m air channel and a 21-m water channel, using benchtop equipment (e.g., benchtop power supplies, arbitrary wave-form generator). It validated the potential of laser light for air-water communication, though assumed a static environment without addressing practical challenges such as scattering and waves. [71] further considered the impact of water height on laser communication to build an adaptive water-air-water link. Assuming a calm water surface, their proposed system does not address issues arising from capillary waves common in real applications. [80] focused on advancing modulation scheme design to mitigate the impact of atmospheric turbulence. Although a water channel was also included as part of the propagation path in the evaluation, no water-related dynamics were addressed in their design. Unlike above works exploring narrow-beam laser light with strict alignment requirements, [63] studied a diffuse optical link with different modulation schemes (e.g., OFDM, QAM) and the impact of waves on the resulting data rates. In comparison, our work considers a narrow and collimated laser beam to save energy while presenting systems solutions to address water wave movement. A recent work [67] combined acoustics and RF to realize the direct link from water to air. Our work differs in that (1) we explore a different medium, (2) our methodology supports bidirectional communication, and (3) we achieve much higher data rates.

Wireless Optical Communication. Wireless optical communication has been well studied in airborne and underwater scenarios individually [45, 62, 79, 81]. The highest reported data rates are 40 Gbps for an airborne link (tested between two buildings $\sim$1 km apart using a laser with an undisclosed optical power [31]) and 12.4 Gbps for an underwater link (evaluated in a water tank up to $\sim$10 m [74] using a 26 mW laser). In water, the attenuation of light (0.39 dB/m) falls between acoustic (0.1–4 dB/km) and RF (3.5–5 dB/m), resulting in a typical communication range of tens of meters [45]. Specifically for robotics applications, [25] demonstrated the use of an LED-based optical link for remote control of their robot. Inspired by these prior works, we establish an optical link across the air-water boundary. We focus on using off-the-shelf hardware to build a portable transmitter and receiver suitable for mobile underwater and aerial robots. Furthermore, we address practical systems challenges specifically related to wave dynamics to enhance link robustness.

9 Conclusion

We presented AmphiLight, a new system framework that enables a bidirectional air-water communication link using laser light. The implemented prototype uses an off-the-shelf MEMS mirror, fisheye lens, LD, and SiPM. Together, with the ultrasonic sensing design, the system performs robustly in real-world experiments. Final results showed throughputs up to 5.04 Mbps at 6.5 m in the air and 2.5 m underwater, making AmphiLight a promising technology to be deployed on flying and underwater drones.

10 Acknowledgments

We sincerely thank reviewers for their insightful feedback and Monika Roznere for her help with the Barbados experiments. This work is supported in part by National Science Foundation (GRFP-1840344, MRI-1919647, CNS-1552924) and the Colby College DavisConnects Research Grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funding agencies or others.

References

[1] Bandpass filter tutorial. https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=10772.

[2] Cree XLamp CXA2520 LED. https://www.cree.com/led-components/media/documents/ds-CXA2520.pdf.

[3] Daylight. https://en.wikipedia.org/wiki/Daylight.

[4] Dispersion. https://en.wikipedia.org/wiki/Dispersion_(water_waves).

[5] Electromagnetic absorption by water. https://en.wikipedia.org/wiki/Electromagnetic_absorption_by_water.

[6] Hc-sr04 ultrasonic sensor distance module. https://www.amazon.com/gp/product/B07H5D43QH/ref=ppx_yo_dt_b_asin_title_o04_s00?ie=UTF8&psc=1.

[7] Laser diodes. https://www.rp-photonics.com/laser_diodes.html.

[8] Ted200c operation manual. https://www.thorlabs.com/thorproduct.cfm?partnumber=TED200C.

[9] Visible laser diodes: Center wavelengths from 404 nm to 690 nm. https://www.thorlabs.com/newgrouppage9.cfm?objectgroup_id=7.

[10] Waterproof ultrasonic module jsn-sr04t. https://www.alibaba.com/product-detail/Waterproof-Ultrasonic-Module-JSN-SR04T-Water_60609536665.html?spm=a2700.wholesale.maylikeexp.1.33b22332i9KFmB.

[11] Where are the highest tides? https://www.co-ops.nos.noaa.gov/faq2.html#26.

[12] Fb520-10 bandpass filter. https://www.thorlabs.com/thorproduct.cfm?partnumber=FB520-10, Accessed 9/19/2019 2019.

[13] Plt5 520b datasheet. https://dammedia.osram.info/media/resource/hires/osram-dam-4960421/PLT5%20520B_EN.pdf, Accessed 9/19/2019 2019.

[14] Sipm working principle. https://www.ketek.net/sipm/technology/working-principle/, Accessed 8/14/2019 2019.

[15] Why use an achromatic lens? https://www.edmundoptics.com/resources/application-notes/optics/why-use-an-achromatic-lens/, Accessed 8/14/2019 2019.

[16] Wld3343 laser diode driver. https://www.teamwavelength.com/product/wld3343-2-2-a-laser-diode-driver/, Accessed 8/14/2019 2019.

[17] The world's fastest 40-nm fpga. https://www.intel.com/content/www/us/en/programmable/products/fpga/features/speed/stxiv-performance.html, Accessed 9/9/2019 2019.

[18] The world's underwater cultural heritage. http://www.unesco.org/new/en/culture/themes/underwater-cultural-heritage/underwater-cultural-heritage/, Accessed 01/20/2019 2019.

[19] Ian F Akyildiz, Dario Pompili, and Tommaso Melodia. Underwater acoustic sensor networks: research challenges. *Ad hoc networks*, 3(3):257–279, 2005.

[20] Alvise Benetazzo. Measurements of short water waves using stereo matched image sequences. *Coastal engineering*, 53(12):1013–1032, 2006.

[21] Yifei Chen, Meiwei Kong, Tariq Ali, Jiongliang Wang, Rohail Sarwar, Jun Han, Chaoyang Guo, Bing Sun, Ning Deng, and Jing Xu. 26 m/5.5 Gbps air-water optical wireless communication based on an OFDM-modulated 520-nm laser diode. *Opt. Express*, 25(13):14760–14765, Jun 2017.

[22] Hyun Choi and Wan-Chin Kim. Design of mechaless LiDAR optical system with large FOV using liquid lens and fisheye lens. In *ASME-JSME 2018 Joint International Conference on Information Storage and Processing Systems and Micromechatronics for Information and Precision Equipment*, page V001T10A001, 2018.

[23] Peter D Craig and Michael L Banner. Modeling wave-enhanced turbulence in the ocean surface layer. *Journal of Physical Oceanography*, 24(12):2546–2559, 1994.

[24] Peng Deng, XiuHua Yuan, Ming Zhao, Yanan Zeng, and Mohsen Kavehrad. Off-axis catadioptric fisheye wide field-of-view optical receiver for free space optical communications. *Optical Engineering*, 51(6):063002, 2012.

[25] M. Doniec, C. Detweiler, I. Vasilescu, and D. Rus. Using optical communication for remote underwater robot operation. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010.

[26] Marek Doniec, Anqi Xu, and Daniela Rus. Robust real-time underwater digital video streaming using optical communication. In *Proc. of ICRA*, pages 5117–5124. IEEE, 2013.

[27] David L Donoho et al. Compressed sensing. *IEEE Transactions on information theory*, 52(4):1289–1306, 2006.

[28] Matthew Dunbabin and Lino Marques. Robots for environmental monitoring: Significant advancements and applications. *IEEE Robotics & Automation Magazine*, 19(1):24–39, 2012.

[29] Yusuf Said Eroglu, Ismail Guvenc, Alphan Sahin, Nezih Pala, and Murat Yuksel. Diversity combining and piezoelectric beam steering for multi-element VLC networks. In *Proceedings of the 3rd Workshop on Visible Light Communication Systems*, pages 25–30, 2016.

[30] N Farr, A Bowen, J Ware, C Pontbriand, and M Tivey. An integrated, underwater optical/acoustic communications system. In *Proc. OCEANS*, pages 1–6, 2010.

[31] Xianglian Feng, Zhihang Wu, Tianshu Wang, Peng Zhang, Xiaoyan Li, Huilin Jiang, Yuwei Su, Hongwei He, Xiaoyan Wang, and Shiming Gao. Experimental demonstration of bidirectional up to 40 gbit/s qpsk coherent free-space optical communication link over 1 km. *Optics Communications*, 410:674–679, 2018.

[32] Randima Fernando. *GPU gems: programming techniques, tips and tricks for real-time graphics.* Pearson Higher Education, 2004.

[33] James D Foley, Foley Dan Van, Andries Van Dam, Steven K Feiner, John F Hughes, J Hughes, and Edward Angel. *Computer graphics: principles and practice*, volume 12110. Addison-Wesley Professional, 1996.

[34] Zabih Ghassemlooy, Wasiu Popoola, and Sujan Rajbhandari. *Optical wireless communications: system and channel modelling with Matlab®*. CRC press, 2019.

[35] Frank Hanson and Mark Lasher. Effects of underwater turbulence on laser beam propagation and coupling into single-mode optical fiber. *Applied optics*, 49(16):3224–3230, 2010.

[36] Alan Harris, James J. Sluss, Hazem H. Refai, and Peter G. LoPresti. Comparison of active beam steering elements and analysis of platform vibrations for various long-range fso links. In *Digital Wireless Communications VII and Space Communication Technologies*, volume 5819, pages 474–485, 2005.

[37] Charley J. Henderson, Brian Robertson, Diego Gil Leyva, Timothy David Wilkinson, Dominic C. O'Brien, and Grahame E. Faulkner. Control of a free-space adaptive optical interconnect using a liquid-crystal spatial light modulator for beam steering. *Optical Engineering*, 44(7):075401, 2005.

[38] V Hilsenstein. Surface reconstruction of water waves using thermographic stereo imaging. In *Image and Vision Computing New Zealand*, volume 2, 2005.

[39] Philip A Hiskett and Robert A Lamb. Underwater optical communications with a single photon-counting system. In *Advanced Photon Counting Techniques VIII*, volume 9114, page 91140P. International Society for Optics and Photonics, 2014.

[40] Ove Hoegh-Guldberg and John F Bruno. The impact of climate change on the world's marine ecosystems. *Science*, 328(5985):1523–1528, 2010.

[41] Sven T.S. Holmstrom, Utku Baran, and Hakan Urey. MEMS laser scanners: a review. *Journal of Microelectromechanical Systems*, 23(2):259–275, 2014.

[42] Ya-Wen Huang, Yuki Sasaki, Yukihiro Harakawa, Edwardo F Fukushima, and Shigeo Hirose. Operation of underwater rescue robot anchor diver iii during the 2011 tohoku earthquake and tsunami. In *Proc. OCEANS*, pages 1–6, 2011.

[43] Slamet Indriyanto and Ian Yosef Matheus Edward. Ultrasonic underwater acoustic modem using frequency shift keying (fsk) modulation. In *2018 4th International Conference on Wireless and Telematics (ICWT)*, pages 1–4. IEEE, 2018.

[44] Mathias Johansson, Sverker Hård, Brian Robertson, Ilias Manolis, Timothy Wilkinson, and William Crossland. Adaptive beam steering implemented in a ferroelectric liquid-crystal spatial-light-modulator free-space, fiber-optic switch. *Applied Optics*, 41(23):4904–4911, 2002.

[45] Hemani Kaushal and Georges Kaddoum. Underwater optical wireless communication. *IEEE access*, 4:1518–1547, 2016.

[46] Yagiz Kaymak, Roberto Rojas-Cessa, Jianghua Feng, Nirwan Ansari, MengChu Zhou, and Tairan Zhang. A survey on acquisition, tracking, and pointing mechanisms for mobile free-space optical communications. *IEEE Communications Surveys & Tutorials*, 20(2):1104–1123, 2018.

[47] Stephanie Kemna, David A Caron, and Gaurav S Sukhatme. Adaptive informative sampling with autonomous underwater vehicles: Acoustic versus surface communications. In *Proc. OCEANS*, pages 1–8, 2016.

[48] Paul C Liu, David J Schwab, and Robert E Jensen. Has wind–wave modeling reached its limit? *Ocean Engineering*, 29(1):81–98, 2002.

[49] Xavier Lurton. *An introduction to underwater acoustics: principles and applications*. Springer Science & Business Media, 2002.

[50] V. Milanović, N. Siu, A. Kasturi, M. Radojičić, and Y. Su. MEMSEye for optical 3D position and orientation measurement. In *MOEMS and Miniaturized Systems X*, volume 7930, page 79300U, 2011.

[51] Veljko Milanovic, Kenneth Castelino, and Daniel T. McCormick. Highly adaptable MEMS-based display with wide projection angle. In *IEEE 20th International Conference on Micro Electro Mechanical Systems*, pages 143–146, 2007.

[52] Hiromasa Oku and Masatoshi Ishikawa. High-speed liquid lens with 2 ms response and 80.3 nm root-mean-square wavefront error. *Applied Physics Letters*, 94(22):221108, 2009.

[53] Melanie Ott et al. Capabilities and reliability of leds and laser diodes. *Internal NASA Parts and Packaging Publication*, 1996.

[54] Arvind Pereira, Hordur Heidarsson, Carl Oberg, David A Caron, Burton Jones, and Gaurav S Sukhatme. A communication framework for cost-effective operation of auvs in coastal regions. In *Proc. FSR*, pages 433–442, 2010.

[55] Jeffery J Puschell, Robert J Giannaris, and Larry Stotts. The autonomous data optical relay experiment: first two way laser communication between an aircraft and submarine. In *Telesystems Conference, 1992. NTC-92., National*, pages 14–27. IEEE, 1992.

[56] Yiming Qian, Yinqiang Zheng, Minglun Gong, and Yee-Hong Yang. Simultaneous 3D reconstruction for water surface and underwater scene. In *Proc. of the European Conference on Computer Vision (ECCV)*, pages 754–770, 2018.

[57] John Rzasa. *Pointing, acquisition, and tracking for directional wireless communications networks*. PhD thesis, 2012.

[58] Wee-Leong Saw, Hazem H. Refai, and James J. Sluss. Free space optical alignment system using GPS. In *Free-Space Laser Communication Technologies XVII*, volume 5712, pages 101–110, 2005.

[59] Amarjeet Singh, Maxim A. Batalin, Michael J. Stealey, Victor Chen, Mark H. Hansen, Thomas C. Harmon, Gaurav S. Sukhatme, and William J. Kaiser. Mobile robot sensing for environmental applications. In *Proc. FSR*, pages 125–135, 2007.

[60] In Keun Son and Shiwen Mao. A survey of free space optical networks. *Digital Communications and Networks*, 3(2):67–77, 2017.

[61] Robert Henry Stewart. *Introduction to physical oceanography*. Texas A & M University College Station, 2008.

[62] Mengnan Sun, Bing Zheng, Lifeng Zhao, Xueqin Zhao, and Feifei Kong. A design of the video transmission based on the underwater laser communication. In *Oceans-St. John's, 2014*, pages 1–4. IEEE, 2014.

[63] Xiaobin Sun, Meiwei Kong, Chao Shen, Chun Hong Kang, Tien Khee Ng, and Boon S Ooi. On the realization of across wavy water-air-interface diffuse-line-of-sight communication based on an ultraviolet emitter. *Optics Express*, 27(14):19635–19649, 2019.

[64] Y. Tashiro, Y. Suito, K. Izumi, K. Yoshida, and T. Tsujimura. Optical system design for laser tracking of free space optics. In *2017 56th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, pages 1098–1103, 2017.

[65] Zhao Tian, Kevin Wright, and Xia Zhou. The darklight rises: visible light communication in the dark. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 2–15, 2016.

[66] Hendrik L Tolman. A mosaic approach to wind wave modeling. *Ocean Modelling*, 25(1-2):35–47, 2008.

[67] Francesco Tonolini and Fadel Adib. Networking across boundaries: enabling wireless communication through the water-air interface. In *Proc. of SIGCOMM*, pages 117–131. ACM, 2018.

[68] Chun-Chin Tsai, Ming-Hung Chen, Yi-Chung Huang, Yi-Cheng Hsu, Yuan-Tsun Lo, Ying-Jyun Lin, Jao-Hwa Kuang, Sheng-Bang Huang, Hung-Lieh Hu, Yeh-I Su, et al. Decay mechanisms of radiation pattern and optical spectrum of high-power led modules in aging test. *IEEE Journal of Selected Topics in Quantum Electronics*, 15(4):1156–1162, 2009.

[69] D. Yu Velikovskiy, V.E. Pozhar, and M.M. Mazur. Acousto-optics devices for high-power laser beam. *WDS'12 Proceedings of Contributed Papers: Part III-Physics*, pages 65–68, 2012.

[70] Lloyd Butler VK5BR. Underwater radio communication. *Originally published in Amateur Radio*, 1987.

[71] Andong Wang, Long Zhu, Yifan Zhao, Shuhui Li, Weichao Lv, Jing Xu, and Jian Wang. Adaptive water-air-water data information transfer using orbital angular momentum. *Optics express*, 26(7):8669–8678, 2018.

[72] Zhiqiang Wang, Lu Lu, Pengfei Zhang, Chunhong Qiao, Jinghui Zhang, Chengyu Fan, and Xiaoling Ji. Laser beam propagation through oceanic turbulence. In *Turbulence and Related Phenomena*. IntechOpen, 2018.

[73] O Wilfert et al. Laser beam attenuation determined by the method of available optical power in turbulent atmosphere. *Journal of Telecommunications and Information Technology*, pages 53–57, 2009.

[74] Tsai-Chen Wu, Yu-Chieh Chi, Huai-Yung Wang, Cheng-Ting Tsai, and Gong-Ru Lin. Blue laser diode enables underwater communication at 12.4 gbps. *Scientific reports*, 7:40480, 2017.

[75] Feng Xiao, Lingjiang Kong, and Jian Chen. Beam-steering efficiency optimization method based on a rapid-search algorithm for liquid crystal optical phased array. *Applied Optics*, 56(16):4585–4590, 2017.

[76] Jinwei Ye, Yu Ji, Feng Li, and Jingyi Yu. Angular domain reconstruction of dynamic 3D fluid surfaces. In *Proc. of CVPR*, 2012.

[77] Liangchen Ye, Gaofei Zhang, Zhen You, and Chi Zhang. A 2D resonant MEMS scanner with an ultra-compact wedge-like multiplied angle amplification for miniature LIDAR application. In *SENSORS, 2016 IEEE*, pages 1–3, 2016.

[78] Zhaoquan Zeng, Shu Fu, Huihui Zhang, Yuhan Dong, and Julian Cheng. A survey of underwater optical wireless communications. *IEEE Communications Surveys & Tutorials*, 19(1):204–238, 2016.

[79] Zhaoquan Zeng, Shu Fu, Huihui Zhang, Yuhan Dong, and Julian Cheng. A survey of underwater optical wireless communications. *IEEE communications surveys & tutorials*, 19(1):204–238, 2017.

[80] Lu Zhang, Han Wang, Xu Zhao, Fang Lu, Xiaoming Zhao, and Xiaopeng Shao. Experimental demonstration of a two-path parallel scheme for m-QAM-OFDM transmission through a turbulent-air-water channel in optical wireless communications. *Optics express*, 27(5):6672–6688, 2019.

[81] Xiaoming Zhu and Joseph M Kahn. Free-space optical communication through atmospheric turbulence channels. *IEEE Transactions on communications*, 50(8):1293–1300, 2002.

[82] Mo Zohrabi, Robert H. Cormack, and Juliet T. Gopinath. Nonmechanical beam steering using tunable lenses. In *2017 Conference on Lasers and Electro-Optics (CLEO)*, pages 1–2, 2017.

[83] Mo Zohrabi, Wei Yang Lim, Robert H. Cormack, Omkar D. Supekar, Victor M. Bright, and Juliet T. Gopinath. Lidar system with nonmechanical electrowetting-based wide-angle beam steering. *Optics Express*, 27(4):4404–4415, 2019.

Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Cloud-Scale Infrastructure

Ze Li[†], Qian Cheng[†], Ken Hsieh[†], Yingnong Dang[†], Peng Huang[*], Pankaj Singh[†]
Xinsheng Yang[†], Qingwei Lin[‡], Youjiang Wu[†], Sebastien Levy[†], Murali Chintalapati[†]

[†]*Microsoft Azure* [*]*Johns Hopkins University* [‡]*Microsoft Research*

Abstract

Modern cloud systems have a vast number of components that continuously undergo updates. Deploying these frequent updates quickly without breaking the system is challenging. In this paper, we present Gandalf, an end-to-end analytics service for safe deployment in a large-scale system infrastructure. Gandalf enables rapid and robust impact assessment of software rollouts to catch bad rollouts before they cause widespread outages. Gandalf monitors and analyzes various fault signals. It will correlate each signal against all the ongoing rollouts using a spatial and temporal correlation algorithm. The core decision logic of Gandalf includes an ensemble ranking algorithm that determines which rollout may have caused the fault signals, and a binary classifier that assesses the impact of the fault signals. The analysis result will decide whether a rollout is safe to proceed or should be stopped.

By using a lambda architecture, Gandalf provides both real-time and long-term deployment monitoring with automated decisions and notifications. Gandalf has been running in production in Microsoft Azure for more than 18 months, serving both data-plane and control-plane components. It achieves 92.4% precision and 100% recall (no high-impact service outages in Azure Compute were caused by bad rollouts) for data-plane rollouts. For control-plane rollouts, Gandalf achieves 94.9% precision and 99.8% recall.

1 Introduction

In a cloud-scale system infrastructure like Microsoft Azure, various teams need to frequently make software changes in code and configurations to deploy new features, fix existing bugs, tune performance, *etc.* With the sheer scale and complexity of such infrastructure, even a small defect in updating one component may lead to widespread failures with significant customer impact such as unavailability of the virtual machine service. Indeed, many catastrophic service outages are caused by some small changes [2, 3, 4, 5, 19].

Each software change, therefore, must be rigorously reviewed and extensively tested. Nevertheless, some bugs could remain uncaught due to the discrepancies between testing and production environment in cluster size, hardware SKU (stock keeping unit), OS/library versions, unpredictable workloads, complex component interactions, *etc.*

Thus, even when a software change passes testing, instead of updating all nodes at once, it is common practice to apply the change to production gradually following a safe deployment policy in the order of stage, canary, pilot, light region, heavier region, half region pairs, other half of region pairs. Figures 1 and 2 respectively show the scope and duration for rollouts in Azure infrastructure. More than 70% of the rollouts target multiple clusters, and more than 20% of the rollouts last for 1,000+ minutes.

Such characteristics imply that the very process of rolling out changes in production during the deployment phase presents an opportunity to catch bad changes in a realistic setting. If faults caused by a deployment can be caught at an early stage, it allows the release manager to stop the bad deployment and roll back the change in time to prevent it from causing broader impact such as a whole-region or worldwide unavailability.

Yet, accurately assessing the impact of a deployment in a cloud system is challenging. Solutions like component-level watchdogs [31] that check for a handful of component-level fault signatures are effective for capturing obvious, immediate issues. They alone, however, are insufficient in catching production issues that are minor locally but severe globally across clusters and/or regions. They may also miss latent issues such as memory leaks that happen hours after a rollout. Additionally, a component-level watchdog may fail to catch issues that arise only when interacting with other components, *e.g.*, cross-components API contract violations.

Besides false negative, false alarm poses another challenge. Figure 3 plots the number of deployments in Azure during a recent three-month window. We can see that hundreds of rollout tasks are happening every day. In addition, transient faults such as service API timeouts and temporary network issues are common in production. All of these events can eas-

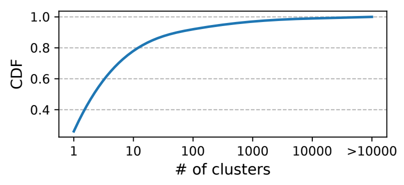


Figure 1: CDF of the scope (number of target clusters) of a rollout.

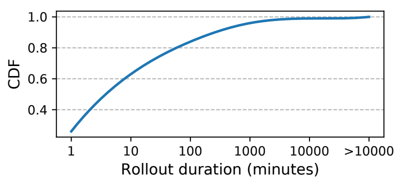


Figure 2: CDF of the rollout duration (in minutes).

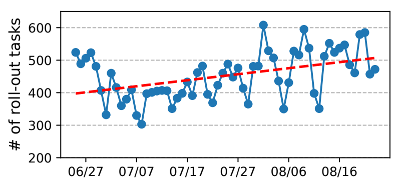


Figure 3: Number of rollout tasks per day. The red dotted line is a trend line.

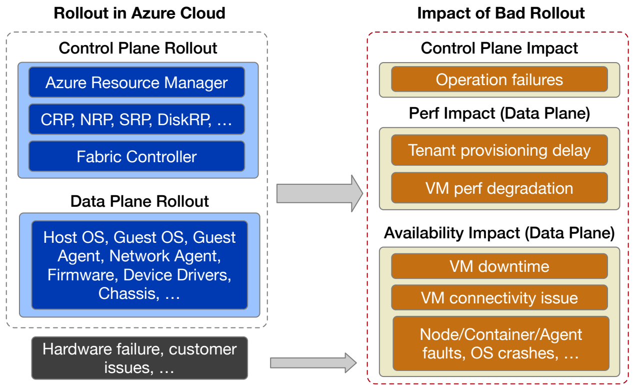
ily mislead a local deployment health monitor to incorrectly attribute a failure to an innocent rollout. These false alarms would cause the innocent rollout to be stopped and prevent timely changes from being applied. Even worse, developers also waste significant time and resources in investigating such false alarms. Consequently, they would not trust future decisions from the monitoring system.

In this paper, we present Gandalf, an end-to-end analytics service that addresses the aforementioned challenges to ensure safe deployment in cloud infrastructure. Instead of analyzing each rollout separately based on individual component logs, Gandalf takes a top-down approach to assess the impact of rollouts holistically. Gandalf continuously monitors a rich set of signals from the infrastructure telemetry data including service-level logs, performance counters, and process-level events. When a system anomaly is detected, Gandalf analyzes if it is caused by a rollout. If a bad rollout is identified, Gandalf makes a "no-go" decision to stop it. Gandalf also provides detailed supporting evidence and an interactive front-end for engineers to understand the issue and the root cause easily.

The core decision logic of Gandalf is a novel model composed of anomaly detection, correlation analysis and impact assessment. The model first detects anomaly from raw telemetry data. It then identifies if a rollout is highly correlated to the detected failures through both temporal and spatial correlation and an ensemble ranking algorithm. Finally, the model uses a Gaussian discriminant classifier to decide if the impact caused by the suspicious rollout is significant enough to stop the deployment.

We design Gandalf system with a lambda architecture [6], combining a real-time decision engine with a batch decision engine. The real-time engine monitors a one-hour time-window before and after the deployment to detect immediate issues; the batch engine analyzes system behavior in a longer time-window (30 days) to detect more complex, latent issues. When Gandalf identifies a bad rollout, it automatically notifies the deployment engine to stop the rollout and fires a ticket with supporting evidence to the owning team.

Gandalf has been running in production for more than 18 months to ensure the safe deployment of Microsoft Azure infrastructure components (*e.g.*, host agents for Compute and Network, host OS). In an 8-month usage window, for data-plane rollouts, Gandalf captured 155 critical failures at the early stage and achieved a precision of 92.4% with 100%



Rollout in Azure Cloud
Control Plane Rollout
Azure Resource Manager
CRP, NRP, SRP, DiskRP, ...
Fabric Controller
Data Plane Rollout
Host OS, Guest OS, Guest Agent, Network Agent, Firmware, Device Drivers, Chassis, ...
Hardware failure, customer issues, ...
Impact of Bad Rollout
Control Plane Impact
Operation failures
Perf Impact (Data Plane)
Tenant provisioning delay
VM perf degradation
Availability Impact (Data Plane)
VM downtime
VM connectivity issue
Node/Container/Agent faults, OS crashes, ...

Figure 4: Different rollouts in Azure and impact of a bad rollout.

recall (meaning that no high-impact incidents, *i.e.*, Sev0-2 outages, were caused by bad rollouts); for control plane, Gandalf achieved 94.9% precision and 99.8% recall, with only two missed issues and two false alarms while monitoring 1200+ region-level deployments. Gandalf has made a significant contribution to getting Azure availability closer to its 99.999% objective by limiting the blast radius of customer VM downtime caused by unsafe rollouts.

Gandalf has also improved the deployment experience for release managers: (1) from looking at scattered evidence to using Gandalf as a single source of truth; (2) from being skeptical about Gandalf decisions to enforcing them; (3) from ad-hoc diagnosis to interactive troubleshooting.

2 Background and Problem Statement

2.1 Deployment in IaaS Cloud

In IaaS cloud, the software stack on the physical nodes and virtual machines (VM) consists of many layers of components. Each component may frequently undergo changes on its independent rollout schedule to add features and fix bugs. These rollouts need to be executed with high velocity and minimum customer impact. For example, in early 2018, Azure quickly deployed a fix to mitigate the Meltdown [30] and Spectre [26] CPU vulnerabilities through a host OS update to keep Azure customers secure. The updates were deployed to millions of nodes that host customer VMs.

As shown in Figure 4, two main kinds of rollouts happen in Azure. The *data-plane* rollouts deploy changes for components running within the hosting environment of customer VMs. For Azure, those components include the host OS, the guest OS, and various software plugins, called agents, inside the host and guest OS. In contrast, *control-plane* rollouts deploy changes to tenant-level services. These services are composed of distributed running instances to manage the system infrastructure and provide interfaces for their functionalities. These control-plane components include the Azure Resource Manager (ARM) [1], which allows customers to query, create, update, and delete VMs with REST APIs and the CRP/NRP/DiskRP (Compute/Network/Disk Resource Provider), which handle customer requests and provision corresponding resources (e.g., virtual disks for VMs). These services are typically structured as loosely-coupled microservices in Azure that communicate with each other via APIs. While loose coupling allows each service to be deployed independently, their deployments also have intricate impacts on each other. Thus, a simple change in one service, while causing no failure in that service, might break the contract with another service and affect a large number of customer API calls if the defective change gets deployed to production.

2.2 Deployment Monitoring System

Requirements To ensure high availability of the VMs and services, rollouts are carefully monitored. Traditionally, safe deployment is a manual process that relies on email communications, multi-party approval, ad-hoc build test validation, and experience-based decisions, which is unsustainable at the scale of Azure. An automated deployment monitoring system is needed to globally oversee rollout progress and automatically stop bad rollout before it causes widespread impact. A deployment monitoring system should detect various anomalies in the large volume of infrastructure telemetry data. In addition, the monitoring system should accurately analyze if the observed failure is caused by a bad deployment or by another issue (e.g., random hardware faults). For the former case, the system should attribute the failure to the responsible deployment among all the ongoing deployments, stop the rollout immediately, and provide supporting evidence for developers to speed up further investigation. For the latter case, the system should not incorrectly blame and stop an innocent deployment.

Target Based on our experience, four kinds of failures happen in production environment: *(1)* hardware issues that happen randomly, *e.g.*, due to firmware bugs, temperature; *(2)* chronic software "hiccups", *e.g.*, due to race conditions; *(3)* hardware-induced outages, *e.g.*, power outage, broken network cable; *(4)* software outage due to bad code or settings in a recent build. Ambient software/hardware issues can be surfaced through separate anomaly detection solutions. In this work, we focus on deployment-related outages, *i.e.*, category *(4)*. Among the four layers of safe-deployment mechanisms

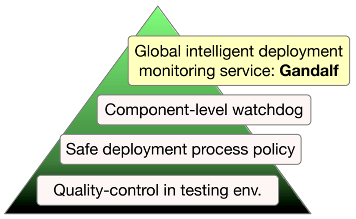

Global intelligent deployment monitoring service: **Gandalf**
Component-level watchdog
Safe deployment process policy
Quality-control in testing env.

Figure 5: Azure's four-layer mechanisms to ensure safe deployment.

in Azure as shown in Figure 5, Gandalf serves as the last safeguard, focus on catching system-level failures, including non-obvious and latent issues, caused by bad deployments.

3 Gandalf System Design

The importance of catching failures during rollouts and the complexity of rollouts in Azure infrastructure motivate the design and implementation of *Gandalf*. Gandalf is an end-to-end, continuous monitoring system for safe deployment. It automates the assessment of rollout impact, the approval or stopping of a rollout, the notification to the owning teams, and the collection of detailed evidence for investigation.

3.1 Design Challenges

In designing Gandalf, we need to address several challenges.

Supporting changes in system and signals. In Azure infrastructure, hundreds of thousands of update events with a large number of fault signals happen every day across the software stack in millions of physical nodes, VMs, and tenants. Gandalf needs to ingest a comprehensive set of data sources and efficiently process them in order to provide timely responses to developers. Furthermore, since Azure extensively employs agile development and micro-service designs, new components emerge and existing components evolve with changing failure patterns and telemetry signals [37]. Gandalf needs to support easy onboarding of new components and telemetry signals while maintaining a robust core decision logic.

Dealing with ambient noise. Ambient faults happen frequently due to diverse reasons such as hardware faults [20], network timeouts, and gray failures [24]. Many of them are unrelated to deployments and can be successfully tolerated. Gandalf needs to deal with such noise. Figure 6 shows an ambient noise example: container faults happened before and after a host OS update on 200+ clusters in Azure; but they were not caused by the host OS deployment—instead, they were caused by firmware defects that prevent the container from starting. User behaviors also affect failure patterns. For example, a customer might invoke a large number of `CreateVM` calls during weekdays. This could lead to an increase of API failures on weekdays and a decrease on weekends. Only monitoring the increase of faults after a deployment could then result in a wrong conclusion.

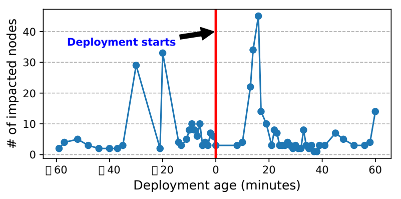


Figure 6: Ambient faults in deployment.

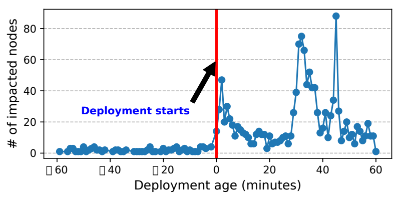


Figure 7: Spike of faults after deployment.

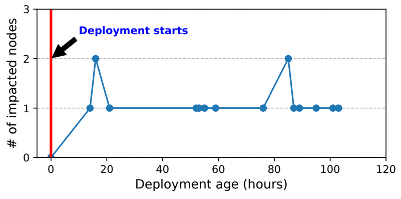


Figure 8: Latent faults after deployment.

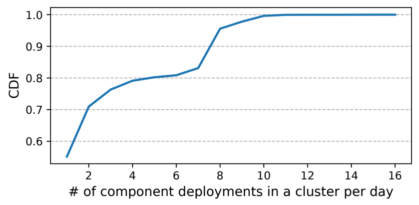


Figure 9: CDF of component count rolled out per cluster in a day.

Balancing speed and coverage. Developers want to get immediate feedback on bad rollouts. Figure 7 shows a typical spike of node faults that happened minutes after a bad rollout. Gandalf needs to quickly alert and stop the bad deployment upon detecting such failures. But a quick decision may be unsound if the failure patterns are subtle. Figure 8 shows an example of a latent issue detected more than 32 hours after the deployment. From the figure, we can see that there is no obvious spike after the deployment of the component. The faults are observed slowly in a long period. Those latent failures are usually only triggered by specific user workloads, e.g., accessing a specific directory or turning on a system service. Therefore, Gandalf needs to detect cross-cluster, latent issues even when a rollout has been ongoing for a while. Covering such latent issues takes longer time by nature.

Identifying the culprit. An n-to-n mapping relationship exists between components and failures: one component may cause multiple types of failure, while a single type of failure may be caused by issues in multiple components. Figuring out which failure is likely caused by which component is not easy due to the complexity of component behaviors. Figure 3 shows that more than 300 deployments take place in Azure every day and the number is increasing. Figure 9 further plots the number of deployment on a cluster per day, showing that about 45% of the clusters have multiple deployments per day.

3.2 System Overview

Figure 10 shows the overview of Gandalf. Gandalf takes a top-down approach in deployment monitoring by consuming telemetry data across clusters for holistic analyses. Gandalf processes three types of data: (1) *performance data* such as CPU performance counters and memory usage in the node; (2) *failure signals* such as agent faults, container faults, OS crashes, node reboots and API call exceptions; (3) *update events* that describe the component deployment information.

In order to balance speed and coverage for safe deployment, the analysis engine of Gandalf is structured in a lambda architecture [6] with a speed layer and a batch layer. The speed layer focuses detects simple, immediate issues and provides quick feedback to developers. The batch layer detects latent, more complex failures and provides more detailed evidence for developers to investigate the issue.

The analysis results from the streaming processed data and batch processed data are consolidated into the serving layer. The serving layer is built as a highly-reliable and scalable web service. The web service stores the analytics results in batch and streaming tables and provides interfaces for various reporting applications to consume the results. The applications include a monitoring front-end for developers to view the rollout status in real-time, a diagnosis UI for investigating problematic rollout, and REST APIs to directly query the result data. Based on the decisions, Gandalf will notify the corresponding component team and create an incident ticket accordingly. Besides notifications, Gandalf publishes the binary decisions for different components into a key/value store to communicate with the deployment engine. The deployment engine subscribes to the signals in the key/value store and stops the rollout if a “no-go” decision is made.

3.3 Data Sources

As a deployment monitoring system, Gandalf continuously ingests deployment events in the Azure infrastructure. These events describe the software build version along with the deployment timestamp and location information. To focus on the system-level impact of rollouts, Gandalf consumes comprehensive signals from various data sources such as service logs, Windows OS events, performance counters, and machine/process/service-level exceptions. Typically each signal is ingested from a separate table in the telemetry data collected in Azure infrastructure. For certain signals, Gandalf performs pre-processing to parse the raw data (e.g., log messages) and extract a failure signature (e.g., an error code). These pre-processed signals are aggregated based on their timestamps, node IDs and service types during the analysis.

When onboarding a new component to Gandalf, the component team needs to provide information about where Gandalf can get the deployment events and the telemetry signals relevant to the component. To ease the correlation analysis

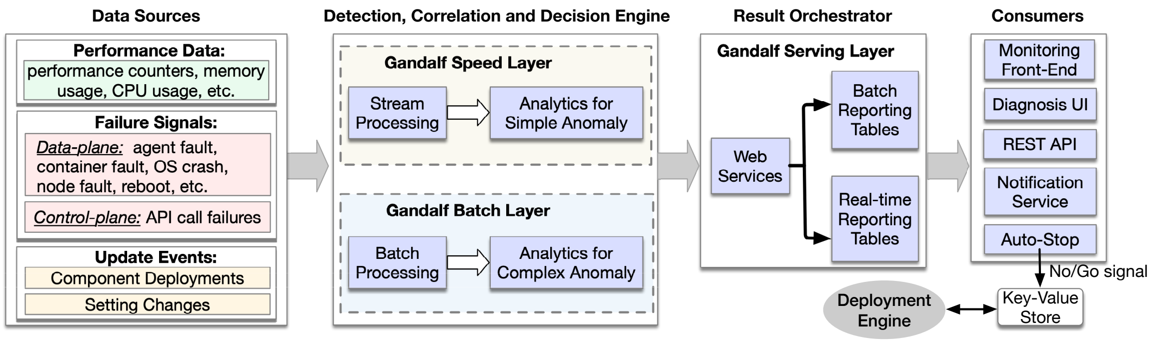


Figure 10: Overview of Gandalf system.

Attribute	Description
Timestamp	When the deployment event completes
Location	Node, cluster, region, etc.
Pivot Group	Hardware SKU, environment, etc.
BuildVersion	Build version identifier
AdditionalInfo	Additional information of the event

Table 1: Gandalf deployment event input data schema.

Attribute	Description
Timestamp	When the fault occurs
Location	Node, cluster, region, etc.
Pivot Group	Hardware SKU, environment, etc.
Signature	Fault signature
AdditionalInfo	Additional diagnosis information

Table 2: Gandalf fault signal input data schema.

between telemetry signals and deployment events, Gandalf requires the information to be structured in a unified data schema as shown in Table 1 and Table 2.

3.4 Stream and Batch Processing

To balance speed and coverage, Gandalf is designed in a lambda architecture [6] with both streaming and batch analysis engines. The speed layer consumes data from a fast pipeline, Microsoft Kusto [7], which is a column-oriented cloud storage supporting analytics with a few minutes of data source delay and up to seconds of query delay. Kusto has a custom query language based on the data-flow model, with native support for streaming operators. Although Kusto has short delays, it cannot efficiently handle a large volume of data using complex algorithms. Therefore, the analysis engine in the speed layer only considers fault signals that happen 1 hour before and after each deployment in each node, and runs lightweight analysis algorithms to provide a rapid response. In Azure, most catastrophic issues happen within 1 hour after the rollout. Latent faults occurring after 1 hour will be captured by the batch layer later.

The Gandalf batch layer consumes data from Cosmos, which is a Hadoop like file system that supports SQL-like query language with up to hours of data source delay. Despite the relatively long delay, Cosmos is an ideal platform for processing an extremely large volume of data using complex models (*e.g.*, it supports external C++ plugins) to detect complex failure and latent issues. This allows the batch layer to analyze faults in a larger time window (30-day period) with advanced algorithms. The lambda architecture allows us to provide both fast decision making and higher coverage over time. A no-go decision can be triggered anytime within the window if the impact scope is large enough.

Both stream and batch analysis in Gandalf are performed in an incremental fashion. Every 5 minutes, the stream processing fetches the latest data streams from Kusto and passes it to the analysis engine in the speed layer. The batch processing runs as an hourly Cosmos job to process the data since the last processing time. The partial results from analyzing each 5 mins mini-batch or hourly batch are aggregated with other partial results in the analysis window to update the overall result. The incremental analysis improves the efficiency as well as the fault tolerance of Gandalf—if the analysis job is restarted, it can resume from the last checkpoint.

3.5 Result Orchestration and Actions

The serving layer of Gandalf is implemented as a highly-reliable and scalable web service using the Azure service fabric framework [8]. After each run of Gandalf's anomaly detection and correlation algorithms (which we will describe in Section 4), the results from the speed and batch layers are stored in two separate reporting tables through the web service. These results contain the deployment impact assessment, the recommended decisions (*"go"* or *"no-go"*), the anomaly patterns, the correlation information, etc.

In general, the telemetry data Gandalf analyzes is both streamed into Kusto and dumped into Cosmos hourly/daily. Given that the data ingested by the speed and batch layer is essentially the same, the reporting results in the two tables are mostly consistent. For example, if the speed layer quickly detects a bad deployment, the batch layer will likely catch it as well, albeit slower. The scenarios when they are inconsistent is mainly when the batch layer reaches a *"no-go"* decision while the speed layer decides a *go*. This is by design since the batch layer makes more informed decisions and covers latent issues that the speed layer cannot detect.

Various DevOps applications pull the results from the reporting tables. This way, the Gandalf system is well integrated into the DevOps workflow. Among the applications, the most important one is the notification service. When the notification service notices a new no-go decision, it sends an email about the decision to the owning team and creates an incident ticket with details. It also notifies the deployment engine via a key-value store to approve or stop the rollout.

3.6 Monitoring and Diagnosis Front-End

Gandalf provides a web front end to enable real-time rollout monitoring and issue diagnosis support for release managers and developers. The feature teams can proactively watch the rollout KPIs (e.g., rollout progress, NodeFaults, Container Faults, OS Crashes, Allocation Failures and etc.) in real-time while waiting for the decision from the Gandalf notification service. After the decision of a rollout is made and sent to the corresponding team, Gandalf provides information to help developers investigate the issue and make a quick fix as needed. For example, Gandalf provides pivot information of the identified issues (*i.e.*, the issue happens only in instances that have specific attributes like SKU).

The Gandalf front-end provides the following views: (i) a binary decision page that summarizes all rollout decisions for different components and build versions in different environments; (ii) a rollout profile page that displays the batch processed decisions and associated diagnosis information; (iii) an issue profile page for a bad rollout with diagnosis information such as the impacted nodes and clusters or the trend in different environments; (iv) a real-time tracking page that shows the rollout progress, related failures, etc.

4 Gandalf Algorithm Design

Existing Algorithms. In designing the algorithms for Gandalf, we considered existing options from supervised learning, anomaly detection and correlation analysis but found major limitations for each of them. Supervised learning is difficult to apply because system behaviors and customer workloads as well as failure patterns and failure-update correlation keep changing. In addition, learning from historical change behaviors does not necessarily help predict the future mapping between failure patterns and new updates. Existing anomaly detection algorithms alone are also insufficient. This is because many rollouts may happen simultaneously in the infrastructure but anomaly detection itself does not tell which deployment is responsible for the anomalies. For correlation analysis, most state-of-the-art methods focus on temporal correlation based on Pearson correlation [34], which cannot capture the complex causal relationship in our scenario.

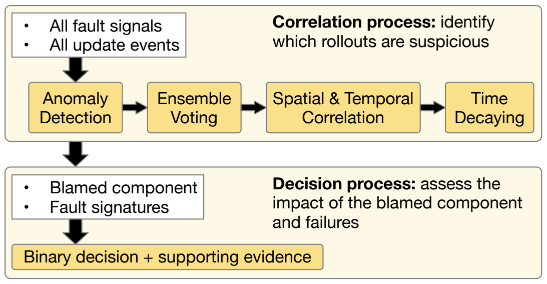

All fault signals
All update events
Correlation process: identify which rollouts are suspicious
Anomaly Detection
Ensemble Voting
Spatial & Temporal Correlation
Time Decaying
Blamed component
Fault signatures
Decision process: assess the impact of the blamed component and failures
Binary decision + supporting evidence

Figure 11: Gandalf correlation model.

Overview. The Gandalf model consists of three main steps: *(1)* anomaly detection detects system-level failures from raw telemetry data; *(2)* correlation analysis identifies the components responsible for the detected failures among multiple rollouts; *(3)* the decision step evaluates the impacted scope and decides whether the rollout should be stopped or not. The correlation step in (2) is further divided into four parts, namely, ensemble voting, temporal correlation, spatial correlation and exponential time decay. Figure 11 shows the overall analysis process in the Gandalf. In the following Section, we describe each step of the algorithm in detail.

4.1 Anomaly Detection

The raw telemetry data that Gandalf analyzes, such as OS events, log messages, and API call statuses, may be imprecise. Therefore, Gandalf first derives concise fault signatures from raw data to distinguish different faults. A raw fault event log usually contains both error codes and error messages. One error code could map to multiple faults if it is too generic. For example, a POST API call that requests compute resources could return `HTTP ERROR 500` for different reasons like `AllocationFailure` or `NetworkExecutionError`. Directly analyzing the error codes would mix different faults, diluting the signal and leading to wrong conclusions. The error messages, on the other hand, are usually non-structured plain text with many unnecessary details. Gandalf processes the raw error messages and applies text clustering [10] to generate fault signatures. We first replace the unique identifiers such as VM ID, subscription ID with dummy identifiers using an empirical log parser similar to prior work [18, 22]. Then we run a simplified incremental hierarchical clustering model [36] to group up all processed text into a set of error patterns, *e.g.*, "Null References" grouped together with `NullReferenceException`.

After obtaining the fault signatures, Gandalf detects anomalies based on the occurrences of each fault signature. Ambient faults, such as hardware and network glitches or gray failures [24], are common in a large-scale cloud system. Simple threshold-based anomaly detection is ineffective because the system and the customer behaviors change over time. With thousands of signatures, it is also unrealistic to manually set thresholds for each. Gandalf instead estimates the baseline from past data using Holt-Winters forecasting [14] to detect anomalies. The training period is set to the past 30 days and the step interval is set to one hour. When the observed value deviates from the expected value by more than 4σ, the point will be marked as an anomaly.

For some components, the occurrences of different fault signatures vary significantly. For example, the volume of client errors could be much higher than platform errors. To better compare the impact of different fault signatures, Gandalf calculates z-score [28], $z_i = \frac{x_i - \mu}{\sigma}$, for each anomaly against historical data in its correlation process.

4.2 Correlation Analysis

A detected failure may not be caused by a bad rollout but other factors such as random firmware issues. In addition, at any point, many concurrent rollout tasks can take place in a large system. Therefore, once anomalies are detected, Gandalf needs to correlate the observed failures with deployment events, and evaluate the impact of the failure on the fly. Note that this identified component might be the triggering component but not necessarily always the root cause.

4.2.1 Ensemble Voting

Since many components are deployed concurrently, we use a vote-veto mechanism to establish the relationship between the faults and the rollout components. For a fault e that happens at timestamp t^f and a rollout component c deployed at t^d on the same node, each fault e votes for all the components deployed before it (i.e., $t^d < t^f$) within a window size w_b and vetoes all the components deployed after it (i.e, $t^d > t^f$) within a window size w_a. Since the deployments are rolled out continuously on different nodes at different time, as shown in Figure 12, we aligned the votes $V(e,c)$ and vetoes $VO(e,c)$ for deployed component c across all nodes based on the fault age as defined as $age(e,c) = t^f - t^d$. P_i are the votes aggregated on WD_i as

$$P_i = \sum_k V(e, c|WD_i), \tag{1}$$

where $age(e,c) < WD_i$. By default, 4 different hour windows are used as WD_1=1, WD_2=24, WD_3=72, WD_4 is the duration between the deployment and the latest data point; k is the number of nodes with the pairs. Similarly, B is the veto aggregated in a single 72 hour window as

$$B = \sum_k VO(e, c|WD_{-1}), \tag{2}$$

where $age(e,c) < WD_{-1}$ and WD_{-1}=72.

4.2.2 Temporal and Spatial Correlation

After ensembling the votes of faults to the components, we calculate the temporal correlation score as

$$ST(e,c) = \sum_{i \in [1,4]} w_i \log\left(\frac{(P_i - B + 1)}{B+1}\right), \tag{3}$$

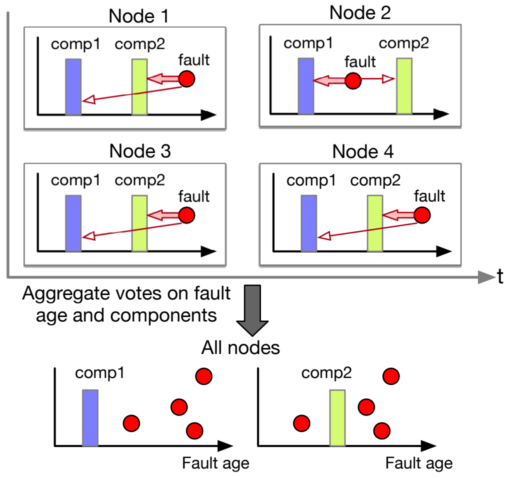


Figure 12: Faults alignment by fault age during ensemble voting. The circle represents a fault. The vertical bar is when a component gets deployed in a node. The arrow blames a fault to a deployment, where the arrow size represents correlation strength.

where $P_i > B$. w_i is the weight of the time window WD_i in Section 4.2.1. This kernel function tries to filter out the ambient faults.

As we do not have ground-truth samples to train the values of w_i in the above equation, we set them empirically. A naïve way would be to assign them the same value. This does not work well in practice because a component deployed closer to a fault usually has a higher correlation, implying the constraint $w_1 > w_2 > w_3 > w_4$. We learned over time that setting *exponential weights* (EW) for w_1 to w_4 works well. The intuition behind exponential weights (EW) is that faults happening right after the deployment are much more likely to have causal relationship than the faults happening a long time after the deployment.

We then evaluate the spatial correlation through

$$SS(e,c|t_1,t_2) = N_f / N_{df}, \tag{4}$$

where N_f is the number of nodes with fault e during the deployment period t_1 to t_2 for component c, and N_{df} represents the total number of nodes with fault e, regardless of whether c was deployed during the same period. If $SS(e,c|t_1,t_2) < \beta$, where β is the confidence level, we will ignore the blaming of the pair. The confidence level can be set as 99% or 90% for different sensitivity. We then identify the blamed component c_j by associating the faults to the component with the largest temporal correlation:

$$blame(e) = \arg\max_{c_j} ST(e, c_j) \tag{5}$$

4.2.3 Time Decaying

The blaming score is calculated based on the fault age as shown in Equation 3. If the same fault signature appears again, the fault may still blame the old rollout if the fault age between the old rollout and the fault signature is smaller. We need to focus on new rollouts and gradually dampen the impact of the old rollout because newly observed faults are less likely to be triggered by the old rollout. In order to achieve this, we apply an exponential time decay factors on the blaming score:

$$blame(e) = blame(e) * (\frac{e^{-t} - e^{-w_s}}{e^{-1} - e^{-w_s}}) * b + a \tag{6}$$

4.3 Decision process

Finally, we make a go/no-go decision for the component c_j by evaluating the impacting scopes of the deployment such as the number of impacted clusters, the number of impacted nodes, number of customers are impacted, *etc.* Instead of setting static thresholds for each feature, the decision criteria are trained dynamically with a Gaussian discriminant classifier [9]. The training data is generated from historical deployment cases with feedback from components teams. Note that the impacting scope feature set used in this step is typically organizational policy oriented so it is stable and independent from software changes or bug fixes. Thus it is feasible to obtain good labels for this learning approach comparing to the input features used in correlation analysis.

4.4 Incorporating Domain Knowledge

Gandalf by default treats the input fault signals equally but also allows developers to specify the importance of certain faults with customizable weights. The weights are relative values ranging from 0 to 100, representing the least to the most importance. The default weight for a fault signal is 1.

Weights are usually adjusted by developers reactively, e.g., after investigating a reported issue. For example, developers for certain service may find out the `TimeoutException` tend to be noisy so they reduce its weight to 0.01 for that service; for the Disk Resource Provider, developers may set the weight of `NullReferenceException` to 10 so that Gandalf becomes more sensitive to this failure signature because it is a strong indicator of a code bug. If the weight is set to 0, this failure signature is whitelisted. For example, since developers know that during the rollout of NodeOSBaseImage, node reboots are expected, the weight for `NodeReboot` can be set to 0 to exclude it from the correlation analysis. In general, developers rarely set a fault weight to 0 to avoid missing true issues unless the signal keeps causing false alarms or to encode special rule like the previous example. Since Gandalf exposes the weight settings to developers through a database table, developers sometimes use scripts to adjust weights in a batch, e.g., lowering weights for all `AllocationFailure*` signatures.

5 Evaluation

Gandalf is a production service in Azure. In this Section, we evaluate its business impact, provide three case studies, and analyze the effectiveness of the core algorithms (Section 4).

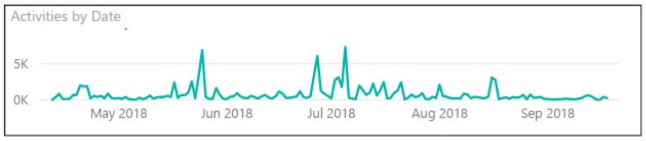


Figure 13: User activities of Gandalf real-time monitoring UI.

5.1 Business Impact

Adoption. The Gandalf service has been running in production and monitoring the Azure infra rollout safety for more than 18 months. It has been widely adopted for the deployments of data-plane components and control-plane services (Section 2.1) across the entire fleet. Specifically, Gandalf currently monitors 19 component rollouts in the data plane including Host OS updates, Agent Package updates, GuestOS updates, *etc.*, as well as 4 control-plane component updates including Compute Resource Provider, Disk Resource Provider, Azure Front End, and Fabric Controller. Figure 13 shows the usage of Gandalf front-end by release managers and developers. Usually, hundreds of page-visits occur every day. When a large rollout happens, the daily number of page visits can reach several thousand.

Scale. The Gandalf system processes on average 270K platform events daily, 770K events on peak days, and logs about 600 million API calls per day in the control plane, including more than 2,000 fault types. The total data volume analyzed is more than 20 TB per day.

Deployment Speed. Gandalf has significantly improved the release velocity. For each deployment, Gandalf can make decisions in about 5 minutes end-to-end on the speed layer, and in about 3 hours on the batch layer. Gandalf cuts the deployment time for the entire production fleet by more than half (Figure 14). As a result, billions of customer API requests will benefit from new features much earlier.

Gandalf streamlines the traditionally cumbersome deployment workflow. Prior to Gandalf, the component-level watchdogs were sometimes noisy or missed important failures. Thus, extensive email communications were needed among the release manager, feature owners and dependent component teams to clear suspicious failure alerts and obtain approval. As a 24/7 monitoring system, Gandalf removes the majority of these costs. Meanwhile, Gandalf provides rich supporting evidence for each alert to facilitate further investigation.

As Azure grows with ever more data to analyze, Gandalf can benefit from additional innovations that improve the performance of analytics systems [29, 35, 38]. For example, if we can cut the delay of our streaming layer from 5 minutes to 10s, it will greatly improve the deployment speed. On the other hand, even with ultra-low data processing latency, Gandalf still needs to wait to accumulate enough evidence for high-confidence decisions. If the rollout quality can be predicted beforehand, the system will have higher business impact. We leave this as our future work.

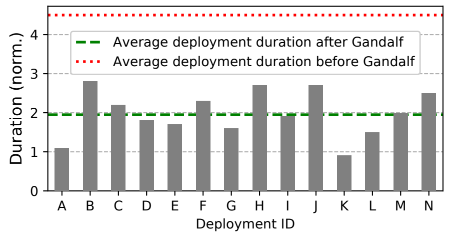


Figure 14: Deployment duration before and after adopting Gandalf

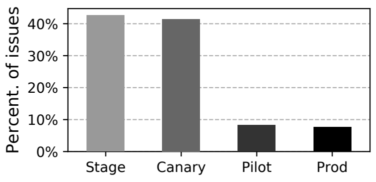


Figure 15: Percentage of issues detected in each environment.

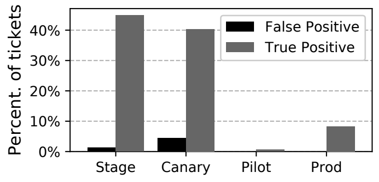


Figure 16: Accuracy of tickets issued by Gandalf in each environment.

5.2 Accurately Preventing Bad Rollouts

Azure enforces a safe deployment policy for all rollouts. Before a component update can be pushed to production, it must pass tests in several environments in the order of Stage, Canary and Pilot. Figure 15 shows the percentage of issues we detected in different environments for rollouts in the data plane. We can see that almost all (99.2%) suspicious rollouts are blocked before reaching production. The rest (0.8%) is blocked in the early stage of production. This means Gandalf effectively limits the blast radius of most bad rollouts.

Once Gandalf stops a rollout, it will send an alert ticket to the corresponding team. Figure 16 shows the accuracy of the alerting in different environments. We can see that most of the false alerting are issued in Stage and Canary environment, which is well aligned to the Gandalf design goal. The false positive rate in Stage and Canary is higher compared to other environments because there are higher levels of noisy failure signals in these environments. Latent issues may mislead the Gandalf service. For example, a faulty agent update accidentally deletes an important folder but does not cause immediate faults. Later, a GuestOS update by customer touched that folder and triggered faults, causing Gandalf to mis-blame the GuestOS deployment.

Overall, in an 8-month usage window from Jan. 2018 to Nov. 2018, Gandalf captured 155 critical failures at the early stage of the data-plane rollouts and achieved a precision of 92.4% with 100% recall (no high-impact incidents were caused by bad rollouts). The detected failures are diverse, including agent faults, OS crashes, node faults, unhealthy containers and VM reboots, which would have caused widespread availability outages. For the control plane, Gandalf has made decisions for 1200+ region-level deployments. The precision is 94.9% and recall is 99.8%. Gandalf filed 39 incidents and only 2 of them are false alarms. Meanwhile, Gandalf automatically approved all other region-level deployments and only missed 2 true issues. One false negative occurred because of incomplete logs. The other was due to the faulty component throwing generic timeout exceptions instead of specific errors, which misled Gandalf.

The most common issues Gandalf caught are compatibility issues and contract breaking issues. Compatibility issues arise when updates are tested in an environment with latest

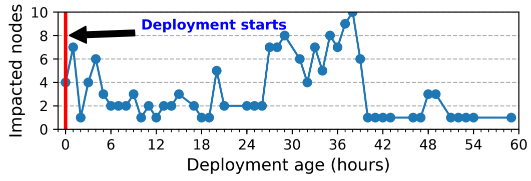


Figure 17: Latent faults caused by network agent deployment.

hardware or software stack but the deployed nodes may have different hardware SKUs or OS or library versions. Contract breaking issues occur when the component does not obey its API specifications and break dependent components.

5.3 Case Studies

We share three representative cases of bad rollouts that Gandalf successfully prevented.

***Case I*: Cross-component Impact.** Release managers tend to ignore faults from other components, which may miss cross-component reliability issues. Gandalf makes informed decision based on anomaly detection and correlation analysis, and is sensitive to such issues. In one case of deploying a Compute Resource Provider (CRP) service update, Fabric Controller (FC) lease failures occurred. When Gandalf first made a no-go decision for the deployment in Canary regions, CRP team claimed these failures were irrelevant to CRP rollout based on their past experience. Therefore, the release manager directly requested to bypass the no-go decision and unblock the rollout. Later on, Gandalf issued another no-go decision in Pilot regions for the same reason, indicating a strong correlation between this failure and CRP rollout. With such evidence, CRP team did a deeper investigation and confirmed it was indeed a regression in CRP. When the customers in Pilot regions reported relevant issues, a hotfix had already been deployed. Because of Gandalf, the regression was caught before it could enter production.

***Case II*: Impact in Specific Region.** After a rollout passes the Pilot regions, release managers typically assume the software updates are in high quality. But Gandalf keeps monitoring throughout the deployment process. Issues that arise at this stage are usually not caused by code bugs of the deployed component but rather the incompatible settings in a specific

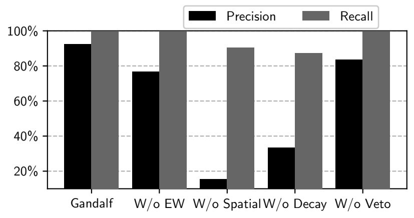


Figure 18: The effectiveness of each Gandalf correlation algorithm.

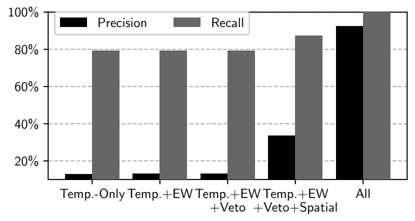


Figure 19: Accumulative effects of correlation algorithms.

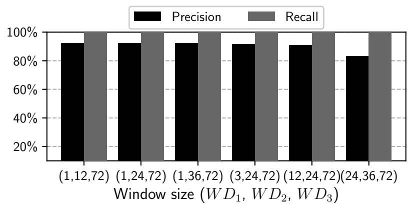


Figure 20: Effectiveness of Window Size.

region. Gandalf is effective in detecting these region-specific issues. For example, during one DiskRP service deployment, Gandalf made a no-go decision in SouthFrance, a late-stage production region. The alert turned out to be caused by a compatibility bug introduced by another component, which is only exposed after the latest DiskRP being deployed in this specific region. With the timely alerting and mitigation, only 3 subscriptions were impacted.

***Case III*: Latent Impact.** Gandalf detects not only immediate issues that happened right after the deployment but also latent issues that happen several hours or even days after deployments. Gandalf detects the latent issues in the early stage and prevents it from affecting customers in production environment. Figure 17 shows a deployment case of a Network Agent in Canary. OS crashed during 24 hours to 72 hours after the deployment and Gandalf issued a Sev2 alert (i.e., large customers impacts). The root cause was a conflict between old firmware version and new driver version. When the network agent rollout upgraded the NIC firmware and drivers, the firmware upgrade script missed one of the hardware revisions. Gandalf accurately attributed the faults to the Network Agent deployment even though the failures occurred 24 hours after the deployment, while many other concurrent updates was ongoing in these clusters.

5.4 Effectiveness of Correlation Algorithms

In this Section, we evaluate the effectiveness of the Gandalf correlation algorithms. The results are from real rollouts in Azure between Jan. 2018 to Nov. 2018.

Parameter Settings. w_i in Equation 3 is respectively set to 8, 4, 2, 1, so that the weights are exponentially decreased along the time windows. The spatial correlation threshold β is set to be 0.8 to tolerate noise in the telemetry data. The w_s in Equation 6 is set to 90 as the longest monitoring period of a rollout is 90 days. b is set to 1 and a is set to 0.2 so that the decay factor is scaled between $[0.2, 1]$.

Exponential Weights. Figure 18 shows that without the exponential weights (EW) used in temporal correlation for different time windows, the precision decreases but the recall remains the same. The reason is that given a spike of faults, without EW, the blame score is high for a bunch of components in addition to the real one, which leads to low precision but the same recall. EW treats component with different fault ages differently, which increase the precision.

Spatial Correlation. Figure 18 shows that by incorporating spatial correlation, Gandalf correlation precision is increased by 77%. The reason is that multiple components are often rolled out in similar time frame in a large system. The temporal correlation results can be very noisy. By incorporating the spatial correlation, the noisy results can be greatly reduced. As the algorithm can accurately identify the problematic components and prevent it from causing high impacts to customers, the recall also increases.

Time Decaying. Figure 18 shows that without the time decaying algorithm, the precision decreases by 58.8% and the recall decreases by 12.7%. The reason is that after a fault (*e.g.*, connection timeout) is detected in a bad rollout, the bug causing the fault will be fixed. Later, the same timeout fault might still occur due to other bugs in another component being deployed. If we do not have the time decaying algorithm, the fault may be still blamed on the old component while missing the faulty new component.

Veto. Gandalf uses a veto mechanism to reduce ambient noise. Figure 18 shows that without the vetos, the precision decreases by 8.7% and the recall rate is the same. The reason is that if the failure signature appears before the rollout, the failure is more likely not related to the rollout. If we only look at the failure after the deployment, we are more likely to stop rollout based on the ambient signals.

Accumulative Effects of Algorithms. Figure 19 shows the accumulative effect of different algorithms. Comparing Figure 19 with Figure 18, we can find that the spatial algorithm and the time decaying algorithm contribute most to precision. Although the individual algorithm such as veto, exponential weights is important as shown in Figure 18, without the spatial correlation algorithm and the time decaying algorithm, their effects alone on precision are relatively small.

5.4.1 Impact of Window Size

Figure 20 shows the effect of window size settings. We can see that the choices of window size do not significantly affect precision. This is because the main purpose of different windows is to differentiate the importance of the time between

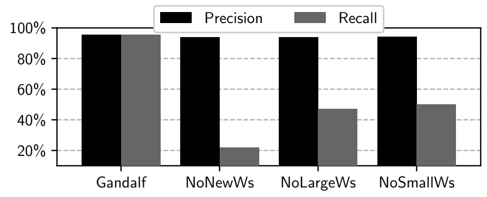


Figure 21: Impact of weight settings for control plane.

updates and anomalies. As long as the window size can provide different weights to the time interval between faults and update events, the precision should be similar as the spatial and temporal correlation algorithms are the major contributor to noise reduction. However, from the figure we can see that if WD_1 and WD_2 windows are too large, the precision decreases. The window size has no effects on the recall due to the same reason as that of the EW effectiveness.

5.4.2 Effectiveness of Weight Adjustments

Figure 21 shows how weight adjustments impact Gandalf decisions on the control-plane rollouts from 01/01/2019 to 03/06/2019. In this experiment, we compare Gandalf decisions using customized weights with 1) decisions without weight adjustment for all new faults (*NoNewWs*); 2) decisions without weight adjustments for all important faults (*NoLargeWs*), *i.e.*, all large weights changed back to 1; 3) decisions without weight adjustment for all noisy faults (*NoSmallWs*), *i.e.*, all small weights changed back to 1. We can see that *NoNewWs* decreases the precision slightly (1.8%) and decreases the recall significantly (73.2%). *NoLargeWs* and *NoSmallWs* have a similar effect on precision and recall. The experiments show that customizing weights can significantly improve recall while maintaining high precision.

From Sep. 2018 to Mar. 2019, 47% of the fault signatures are assigned with non-default weights, with 5 to 10 weights customized for a typical component/team. During this period, only the weights of 18 signatures in total are adjusted 4 times by developers. Thus, the tuning efforts overall are small.

6 Discussion

Gandalf has been running in Azure production environment for more than 18 months. In this section, we share perspectives from our users (Azure engineers and release managers) and lessons we learned along the way.

6.1 Transforming Deployment Experience

"We can call it a very good day for Gandalf!"

"This is a good case for Gandalf and a lesson or two for us"

"Gandalf has helped our rollout to the better. Thanks!"

– Comments from our users

The impact of Gandalf goes beyond accurately preventing bad rollouts. We are thrilled to witness how Gandalf has transformed the engineers and release managers' experience in deploying software changes:

From looking for scattered evidence to using a single source of truth. Before Gandalf was created, component-level watchdogs are used for safe deployment. These watchdogs only have isolated views about individual components. It is therefore difficult to rely on them for safe deployment. Consequently, release managers still need manual efforts to check the deployment behavior from additional data sources and communicate across related teams for ensuring deployment safety. The additional communication cost and decision overhead caused the deployment period to be long. Gandalf ingests comprehensive data sources in the data plane and control plane, and runs system-level analysis with anomaly detection and correlation models. Therefore, Gandalf can provide a single source of truth with various dimensions.

From skeptic to advocate. When some teams adopt Gandalf, the experienced engineers may be initially skeptical about Gandalf's data-driven approach and its decisions. As Gandalf detects complex failures that even experts can miss, the engineers start to trust Gandalf decisions and enforce the team to carefully investigate each "no-go" alert by Gandalf. For many teams, the deployment policy has become that the rollout will not continue to the next region unless Gandalf gives a green light decision.

From ad-hoc diagnosis to interactive troubleshooting. Before Gandalf, when an alert was sent to a component team, the engineers needed to write various queries for multiple data sources or access some sample nodes to grep the fault traces for diagnosis. Gandalf provides an interactive diagnosis portal to directly show the fault details. In particular, Gandalf aggregates the VM-level, node-level and cluster-level faults and buckets these faults based on their fault types. For example, faults "`Failed function: RuntimeVmBaseContainer::ValideXBlockBaseDisk: 0x81700035, XDiskLeaseIdMismatchWithBlobOperation: 0xc1425034`" will be bucketed into `ContainerFault-Creation-DiskLeaseIdMismatch`. Developers can drill down each fault bucket interactively to inspect the detailed fault source such as error messages and logs. The portal also shows the historical baselines to illustrate the difference so that the developers can better understand the impact scope and severity. Moreover, the portal shows the pivot analysis results, *e.g.*, SKU Gen2.3, that highlight potential causes.

6.2 Lessons Learned

We also learned several lessons while we built Gandalf. First, while F-score is a widely used metric to balance the precision and recall of a decision model, in reality, different components may favor precision and recall differently. For teams with limited engineering capacity, they often prefer a system that only sends true alerts so that engineers can focus on investigating true issues. For teams that manage mission-critical services, 100% recall is a strict requirement. Missing any true issues causes much more damage than false alarms. A monitoring system should be tailored for different needs. For example, to fix the Meltdown and Spectre CPU vulnerability, the updates needed to be deployed to millions of nodes quickly. Since the rollout would impact millions of customers, Gandalf was optimized for extremely high recall and feature teams used the interactive portal to proactively monitor the rollout. False alarms were less critical in this scenario as engineer resources were enough to investigate all Gandalf reported issues.

Second, transparency and supporting evidence are crucial to build trust. It is difficult to trust machine decisions, especially on critical tasks. In cloud deployments, the release manager holds the same opinion because a simple false decision could be extremely harmful. That is why a black-box service that is not explainable is hard to be adopted for deployment monitoring even if the decisions are highly accurate. To gain trust, we design the Gandalf model to match the human decision process and make every step transparent. Gandalf surfaces rich supporting evidence, including the ranked list of faults, where the faults occurred, comparison of the time-series signal data before and after deployment, statistical summaries of the impact scope (*e.g.*, how many nodes and customers are affected). Such evidence helps explain to release managers why Gandalf makes each decision.

Third, analytics models should be adaptive. Many standard anomaly detection and time series algorithms are ineffective in a large-scale production system if applied without domain knowledge. It is almost impossible or at least extremely costly to learn such domain knowledge purely from the data. This is especially true when the system is constantly evolving, *e.g.*, an increasing number of new fault signals will emerge. We work closely with engineers to continuously incorporate their input into Gandalf decision model (because domain knowledge may not be fully discovered in one shot!).

7 Related Work

Time-series based anomaly detection models [13] provide high-quality alerts in DevOps. Instead of checking raw logs, DevOps can focus on the anomalous failure events while new build is rolling out [39]. Hangal and Lam first introduce DIDUCE [21], a practical tool that detects complex program errors and identifies root causes. Wang *et al.* [40] propose entropy-based anomaly testing, which uses arbitrary metrics distributions instead of fixed thresholds, for online systems anomaly detection. Fu *et al.* [17] propose classification algorithms to identify performance issue beacons. Laptev *et al.* [27] design a generic time-series anomaly detection framework, EGADS for Yahoo. Cohen*et al.* [16] use Tree-Augmented Naive Bayes models (TAN) to correlate SLO with system states as signatures. Panorama [23] detects gray failures through instrumenting observability hooks in the source code of observer components. However, without correlating anomalies with operational events, these work cannot identify which rollout is responsible or whether the detected anomalies are unrelated to deployments.

A number of tools have been built to analyze the correlation between KPI signals and system state changes. Bahl *et al.* [11, 12, 15] propose an inference graph that captures the dependencies between all components of the IT infrastructure by combining together these individual views of dependency and tries to locate the root cause. Azure is growing so fast that it is hard to build such dependency graph accurately at low cost. Several other methods are proposed for correlating systems signals and events [25, 32, 33, 41, 42]. But they mainly focus on extracting correlations in temporal dimension. Pure temporal correlation is insufficient for accurately identifying bad rollouts in our scenario.

8 Conclusion

In cloud infrastructures that undergo frequent changes, ensuring bad rollouts are accurately caught at the early stage is crucial to prevent catastrophic service outage and customer impact. In this paper, we present Gandalf, an end-to-end analytics service for safe deployment of cloud infrastructure. Gandalf assesses system-level impact of deployments by designing anomaly detection, correlation analysis and failure impact analysis algorithms in its decision model. It uses a lambda architecture to provide both real-time and batch deployment monitoring, with automated deployment decisions, a notification service and a diagnosis front-end. Gandalf has been running in Azure production for more than 18 months. Gandalf blocked 99.2% of the bad rollouts before they enter production. For data-plane rollouts, Gandalf achieved 92.4% precision with 100% recall. For control-plane rollouts, Gandalf achieved 94.9% precision and 99.8% recall.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd, Sujata Banerjee, for their valuable comments. We thank Microsoft Azure engineers who have been closely working with us, giving us feedback, adopting and using the Gandalf system, especially Chango Valtchev, Pengfei Huang, Alp Onalan, Francis David, Naga Govindaraju, Richard Russo, Roy Wang, Huaming Huang, Abhishek Kumar, Yue Zhao, Binit Mishra, Rishabh Tewari, Raza Naqvi, Igal Figlin, Anupama Vedapuri, and Cristina del Amo Casado. We thank Girish Bablani, Mark Russinovich, Dongmei Zhang, and Marcus Fontoura for their great support and sponsorship.

References

[1] Azure resource manager overview. https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview.

[2] Facebook 2.5-hour global outage. https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919, 2010.

[3] Update on Azure storage service interruption. https://azure.microsoft.com/en-us/blog/update-on-azure-storage-service-interruption/, 2014.

[4] Google Compute Engine all-region downtime incident #16007. https://status.cloud.google.com/incident/compute/16007?post-mortem, 2016.

[5] Amazon Web Service suffers major outage, disrupts east coast internet. http://www.datacenterdynamics.com/content-tracks/colo-cloud/aws-suffers-a-five-hour-outage-in-the-us/94841.fullarticle, 2017.

[6] Lambda architecture. http://lambda-architecture.net/, 2017.

[7] Getting started with kusto. https://docs.microsoft.com/en-us/azure/kusto/concepts/, 2018.

[8] Service fabric. https://azure.microsoft.com/en-us/services/service-fabric/, 2018.

[9] H. Abdi. Discriminant correspondence analysis. In *Encyclopedia of Measurement and Statistic*, 2007.

[10] C. Aggarwal and C. Zhai. *Mining Text Data*, chapter A Survey of Text Clustering Algorithms, pages 77–128. Springer, 2012.

[11] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 13–24, Kyoto, Japan, 2007.

[12] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI'04, pages 18–18, San Francisco, CA, USA, 2004.

[13] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[14] C. Chatfield. The Holt-Winters forecasting procedure. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 27(3):264–279, 1978.

[15] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, E. Brewer, E. Brewer, and E. Brewer. Path-based faliure and evolution management. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, NSDI'04, pages 23–23, San Francisco, CA, USA, 2004.

[16] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 105–118, Brighton, United Kingdom, 2005.

[17] Q. Fu, J.-G. Lou, Q.-W. Lin, R. Ding, D. Zhang, Z. Ye, and T. Xie. Performance issue diagnosis for online service systems. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, pages 273–278, Irvine, CA, USA, 2012.

[18] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 149–158, Miami, FL, USA, 2009.

[19] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 1–16, Santa Clara, CA, USA, 2016.

[20] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST '18, pages 1–14, Oakland, CA, USA, 2018.

[21] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, Orlando, FL, USA, 2002.

[22] S. He, Q. Lin, J.-G. Lou, H. Zhang, M. R. Lyu, and D. Zhang. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 60–70, Lake Buena Vista, FL, USA, 2018.

[23] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16. USENIX Association, October 2018.

[24] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, Whistler, BC, Canada, May 2017.

[25] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 243–254, Barcelona, Spain, 2009.

[26] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[27] N. Laptev, S. Amizadeh, and I. Flint. Generic and scalable framework for automated time-series anomaly detection. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1939–1947, Sydney, NSW, Australia, 2015.

[28] D. N. Lawley. A generalization of Fisher's z test. *Biometrika*, 30(1/2):180–187, 1938.

[29] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. STREAMSCOPE: Continuous reliable distributed processing of big data streams. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 439–453, Santa Clara, CA, USA, 2016.

[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[31] C. Lou, P. Huang, and S. Smith. Comprehensive and efficient runtime checking in system software through watchdogs. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS '19, Bertinoro, Italy, May 2019.

[32] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu. Mining program workflow from interleaved traces. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 613–622, Washington, DC, USA, 2010.

[33] C. Luo, J.-G. Lou, Q. Lin, Q. Fu, R. Ding, D. Zhang, and Z. Wang. Correlating events with time series for incident diagnosis. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1583–1592, New York, New York, USA, 2014.

[34] K. Pearson. Notes on regression and inheritance in the case of two parents. In *Proceedings of the Royal Society of London*, 1895.

[35] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 275–288, Seattle, WA, USA, 2014.

[36] N. Sahoo, J. Callan, R. Krishnan, G. Duncan, and R. Padman. Incremental hierarchical clustering of text documents. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, pages 357–366, Arlington, VA, USA, 2006.

[37] T. Schlossnagle. Monitoring in a DevOps world. *Communications of the ACM*, 61(3):58–61, Feb. 2018.

[38] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, Shanghai, China, 2017.

[39] J. Waller, N. C. Ehmke, and W. Hasselbring. Including performance benchmarks into continuous integration to enable DevOps. *SIGSOFT Softw. Eng. Notes*, 40(2):1–4, Apr. 2015.

[40] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *2010 IEEE Network Operations and Management Symposium*, NOMS '10, 2010.

[41] S. Zhang, Y. Liu, D. Pei, Y. Chen, X. Qu, S. Tao, Z. Zang, X. Jing, and M. Feng. Funnel: Assessing software changes in web-based services. *IEEE Transactions on Services Computing*, Volume: 11:34–48, 2016.

[42] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 358–369, Hong Kong, China, 2002.

Experiences with Modeling Network Topologies at Multiple Levels of Abstraction

Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley
Google LLC, Mountain View, CA

Xiaoxue Zhao
Alibaba Group Inc.

Abstract

Network management is becoming increasingly automated, and automation depends on detailed, explicit representations of data about the state of a network and about an operator's intent for its networks. In particular, we must explicitly represent the desired and actual topology of a network. Almost all other network-management data either derives from its topology, constrains how to use a topology, or associates resources (e.g., addresses) with specific places in a topology.

MALT, a Multi-Abstraction-Layer Topology representation, supports virtually all network management phases: design, deployment, configuration, operation, measurement, and analysis. MALT provides interoperability across our network-management software, and its support for abstraction allows us to explicitly tie low-level network elements to high-level design intent. MALT supports a declarative style, simplifying what-if analysis and testbed support.

We also describe the software base that supports efficient use of MALT, as well as numerous, sometimes painful lessons we have learned about curating the taxonomy for a comprehensive, and evolving, representation for topology.

1 Introduction

As our networks get bigger and more complex, we must automate all phases of network management. Automation depends on precise and accurate representations of the desired and actual network. While network management requires many categories of data, the most central is a representation of desired or actual network topology. Almost all other network-management data either derives from topology, or provides policy for how we want to create and use the topology, or associates resources (such as IP addresses or hardware inventory) with specific places in a topology.

At Google, we have learned the value of driving our network management processes, from capacity planning through network design, deployment, configuration, operation, and measurement, using a common standard representation of topology – a "model." Such a representation needs to address multiple problems, including:

- Some management processes, e.g., capacity planning, need to operate on abstractions, such as the amount of future capacity between two cities (before we know how that capacity is implemented). Other processes, e.g., configuration for routers, or fault localization, must operate on low-level details (fibers, switches, interfaces, SDN controllers, racks, connectors, etc.). Still other processes, e.g., risk analysis, must reason about dependencies between abstract and physical concepts. Therefore, we must **represent multiple levels of abstraction, and the relationships between them**.
- In our experience, it is impossible for an unchanging schema to successfully represent a constantly-changing network, especially with frequent technical innovations. Therefore, the representation must support **extensibility and evolution** as first-class features.
- We must support constant change to our network, with many concurrent changes happening at once. We must also support "what-if" analyses of options for future topologies. While humans often prefer an imperative style ("add this", "move that", "change that"), we have found that using versioned sequences of **declarative** models removes a lot of complexity created by imperative operations. For example, with a declarative model we can validate the consistency of an entire network before committing a change.
- Different parts of our network follow different design and operational styles, managed by different teams. By **sharding** our models on carefully-chosen boundaries, we enable these teams to use their preferred styles without excessive interference. Sharding also improves concurrency. At the same time, these shards do overlap, and we prefer, when possible, to use tools that work across our entire network, so a **uniform representation** ensures interoperability.

This paper describes MALT (for *Multi-Abstraction-Layer Topology*), the representation we have developed for network topology, the software and processes we have built around it, and the lessons we have learned. MALT's primary goal is to provide a single representation suitable for almost all use-cases related to network topology. Our path from "we need a uniform topology representation; how hard could that be?" to our current ecosystem has exposed that creating a coherent set of design decisions, and getting this to work at scale, was much harder than we expected[1].

The main contributions of this paper vs. prior work are to explain the value of a multi-abstraction-layer representation that supports the full lifecycle of a network, and to expose some pitfalls that await designers of similar representations.

2 Uses for MALT

We model our global WAN [10], cloud-scale datacenter [20], and office-campus networks using MALT, including both Software-Defined Network (SDN) and traditional network designs. Over 100 teams in Google and hundreds of engineers and operators use MALT regularly; we now require most systems working with network topology to use MALT.

Our uses have expanded over the past five years, and continue to expand to new phases of network lifecycles, and to new network types. A common theme across these uses is that they enable automation of diverse and interacting management processes, through a uniform representation of the intent for, and structure of, our current and future network.

Broadly, our primary uses for MALT have been in three areas: operating our user-facing WAN; WAN capacity planning and design; and datacenter fabric design and operation.

Operational management of user-facing WAN: We use MALT when configuring network elements, managing the control plane of an in-service network, and monitoring the network for failures and other behaviors. We represent the network's "as-built" state in a MALT model in which all entities related to device management, including routers, interfaces, links, Points of Presence (POPs), etc, are visible to management workflows, which only operate by updating this model, and never by directly updating devices. An API supports specific, well-defined update operations on this model; all updates are validated before being used to generate the corresponding device configurations. Operations include adding devices or links, or changing device (entity) attributes to control monitoring and alerting.

WAN capacity planning and design: We must explore many options for evolving network capacity to meet predicted demands. MALT's support for multiple layers of abstraction, including layer-1 elements (e.g., fiber cables and transponders) and layer-3 elements (routers, line cards, etc.), allows simulation of each option against specific failure models, so that we can jointly optimize failure resilience and network cost.

[1]While many enterprise networks are smaller than Google's, we believe that MALT's approach would also be beneficial at much smaller scales.

Consider an example where our demand forecast suggests adding capacity between two POPs. We construct *candidate* MALT models for each possible option for long-haul fiber spans, optical line system elements, and available router ports in those POPs. Then we compare options for incremental cost, lead times for hardware or fibers, and availability. We commit to one option, which becomes the *planned* MALT model, used to generate a detailed design and bill of materials which is consumed by deployment teams.

Topology design for datacenter fabrics: Web-scale datacenter fabrics are far too large for us to directly manage each individual device. Our fabrics are structured as abstract blocks of switches [20]. We use MALT to describe the fabric in terms of these abstract blocks, their sizes, and policies governing how they should be interconnected. This abstraction makes it possible to reason about the topology. Once the complete high-level design is determined, the abstract topology is transformed mechanically into a fully concretized fabric model, also represented in MALT, from which device configurations and other management artifacts can be generated. We maintain both the abstract and concrete representations in MALT, to enable correlation between a given device and the abstract entity that generated it.

While we have many uses for MALT, our network management processes often use data that is *not* about topology, and for which we use other abstract or concrete representations; § 8 discusses what we exclude from MALT.

2.1 Motivations for MALT

The use cases above illustrate three motivations for MALT:

- **Support for the full lifecycle of a network**: As described above, we use MALT models for capacity planning, reliability analysis, high-level and detailed design, deployment planning and auditing, and as one kind of input to the generation of device and SDN-controller configuration. MALT also supports our monitoring, bandwidth management, and some debugging operations. We regularly find new uses, often without having to make major schema changes.
- **Uniformity**: Prior to adopting MALT, we had many systems that maintained representations of network topology for their own uses. These systems often have to exchange topology data. Without a single, uniform representation, this leads not only to O(N^2) translations, but also the potential for data loss or ambiguity, both of which make real automation nearly impossible. Uniformity also allows hundreds of software engineers to write interoperable systems without massive coordination overheads (which also tend to grow as N^2).
- **Multiple levels of abstraction**: Many design, operation, repair, and analysis processes require a clear understanding of the relationships between high-level design intent

and low-level realizations. For example, when we analyze a WAN plan to understand whether it will meet its availability SLO [1], we need to know the physical locations of the underlying fibers – e.g., whether two fibers run across the same bridge or under the same cornfield. MALT allows us to explicitly represent these abstraction relationships (see §3.3), which allows software to operate on data, rather than relying on inference.

2.2 Support for the entire network lifecycle

We present some illustrations of how we could use MALT models for various points in the lifecycle of our networks.

Consider a WAN with 10gbps capacity between London and Paris, which want to increase to 20gbps. Such increments often take months, so we start by creating a model ① (see Fig. 1) of our WAN for (say) 6 months from now, with this capacity set to 20gbps. We then look for available submarine and terrestrial cable capacity that collectively provides an additional 10gbps between the two cities. There might be several possible cable paths, so we can create "what-if" ② models for each option, and then analyze each option with respect to costs, lead-times, and historical failure probabilities, before committing to a "plan of record" (POR) model ③ for the WAN connectivity.

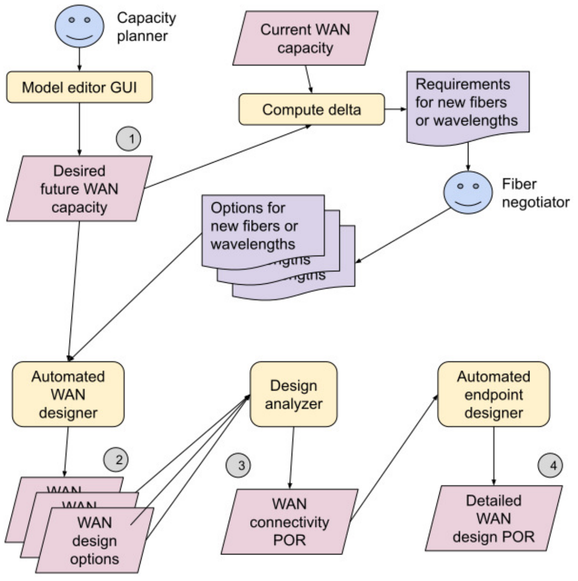


Figure 1: Multiple MALT models for WAN planning

Then we must choose endpoint equipment (routers, optical line systems, etc.) and either ensure we have enough free ports on existing equipment, or order new systems; again, we often explore multiple options (different vendors, different router configurations, etc.) before choosing a final POR model ④ that covers both WAN links and terminating equipment. (This also includes ensuring that routers can physically fit into available racks, and that we have enough power and cooling.)

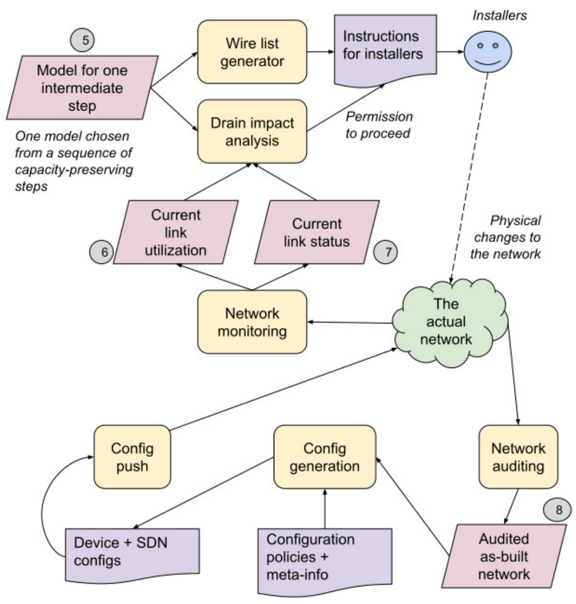


Figure 2: Multiple MALT models for a capacity expansion

Now consider a datacenter capacity expansion (see Fig. 2). Because we expand live networks [23], we must split up the physical changes into multiple steps, to maintain capacity headroom – e.g., an 8-step procedure should reduce capacity by at most 12.5% per step. For each step, we generate an intermediate model ⑤, which we subject to an an automated "drain-impact analysis" just before executing the step. This analysis uses two additional models: the current link utilizations ⑥ to estimate near-term future demands, and a model representing failed links ⑦ to account for their impact on available capacity.[2]

Once a step is ready, we trigger the human operations to carry it out. Humans are fallible, and so is hardware, so before "un-draining" the new links, we audit the work, via both automated and manual checks. In some cases, we might decide that an error (e.g., using the wrong patch-panel port) is harmless, but to avoid further confusion, we update a model ⑧ of the "as-built" network, so that future changes will not conflict with reality.

After the new hardware is deployed, we must update router configurations. Rather than having humans do that, we auto-generate all such "config" from MALT models and other meta-information [12]. We also use these models to auto-generate configurations for our SDN control plane.

While many of the processes in Figs. 1 and 2 operate on the entire graph of a WAN or datacenter network, others use a query API (§ 6). For example, the wire-list generator (Fig. 2) can query to extract just a set of switch ports and their locations, while ignoring most of a model.

Automation: We asserted a goal of ubiquitous automation. In fact, we have not yet automated every step shown in Fig. 1,

[2]In practice, we merge ⑥ and ⑦ into a single input, also critical to our Bandwidth Enforcer system for online WAN-bandwidth allocation [13].

but most of Fig. 2 now uses MALT-based automation. Full automation requires disciplined, hard work; MALT's goal is to enable that work, not to make it happen by magic.

2.3 Antecedents to MALT

Our efforts to model network topology started with various independently-developed, non-interoperable representations – not through a conscious decision, but because each of multiple teams realized they needed topology modeling. E.g., we had one way to represent datacenter and B4 WAN [10] network *designs* (Fig. 2, ⑤), and an entirely different representation, to support bandwidth allocation [13], for link *status and utilization* (Fig. 2, ⑥+⑦); the necessary format conversion was hard to maintain. Other teams maintained database-style records for each WAN router, but resorted to spreadsheets or diagrams to represent WAN topology, without machine-readable abstractions tying capacity intent (Fig. 1, ①) to specific links (Fig. 1, ④).

The lack of abstraction and interoperability between these formats created significant complexity for our operations and software. While MALT has not entirely eliminated that complexity, it gives us a clear path.

3 The MALT representation

We chose to use an "entity-relationship model" representation for MALT. (In § 5.3 we explain why we chose not to expose a relational database). In an entity-relationship model, entities represent "things," which have have "kinds" (types), names, and attributes, Entities are connected via relationships, which (in MALT) have kinds, but neither names nor attributes. MALT uses a somewhat simplified form of the classic entity-relationship model [4].

Our current schema has O(250) entity-kinds, including (among many other things) **data-plane elements**, such as packet switches, switch ports, links between ports, etc.; **control-plane elements**, such as SDN controller applications, switch-stack "control points" (local control planes), machines for SDN controllers, etc. **"Sheet-metal-plane" elements**, such as racks, chassis, and line-cards. Designing, curating, and evolving this taxonomy has been challenging; we discuss our experiences in § 9.

We have a set of about 20 relationship-kinds, including "contains" (e.g., a line card contains a packet-switch chip), "controls" (e.g., an SDN controller application controls a switch's local control plane), and "originates" (e.g., a link originates at one port, and terminates at another).

By convention, we name entity-kinds such as `EK_PACKET_SWITCH` and `EK_RACK`, and relationship-kinds such as `RK_CONTAINS` and `RK_ORIGINATES`.

For each entity-kind, we define a set of attributes. Some entity-kinds have lots of attributes, some have only a few. Typical attributes include:

- the "state" of an entity: is it planned? deployed? configured? operational? faulty? under repair? etc. (We defined a uniform "state machine," although adopting this standard ubiquitously has been challenging.)
- the index of an entity within its parent (e.g., "this is linecard #3 within the containing chassis").
- IP addresses and VLAN IDs assigned to interfaces – useful when generating device configuration.
- the maximum capacity (bits per second) for elements such as ports and links – useful for capacity-planning and traffic engineering.

but attributes can be rather arcane, such as one meaning "during the transition from IPv4 to IPv6, this network requires hosts, rather than switches, to perform 6to4 decapsulation."

A complete MALT "model" consists of a set of entities and a set of relationships between those entities. A model also includes some metadata, such as the provenance of the model (what software created it, when, and from what inputs) and its profile(s) (§3.5).

Fig. 3 shows a trivial example of a MALT entity-relationship (E-R) graph, depicting a connection between two routers. Each router contains one L3 interface and an L2 "port" for that interface. The interfaces are connected by a pair of unidirectional L3 "logical packet links" that each traverses, in this case, a single L2 "physical packet link." (Link entities in MALT are always unidirectional, which means that they usually come in pairs.)

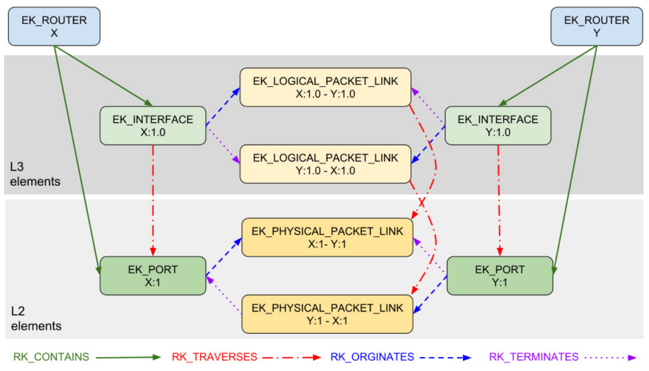


Figure 3: Trivial MALT entity-relationship graph

Note that the E-R graph is not isomorphic to the network graph – links in the network are represented as nodes (entities) in MALT's E-R graph.

Appendix A provides a more detailed example, showing how we model a datacenter network.

3.1 Entity-IDs

MALT entities have *entity-IDs* composed of an entity-kind and an entity-name. E.g., in Fig. 3, one router has an entity-ID of `EK_DEVICE/X` and one link's ID is `EK_PHYSICAL_PACKET_LINK/X:1-Y:1`. Entity-IDs must be globally unique, with respect to an implicit namespace that (by default) covers all of Google, and within a single "snapshot" view of our modeling data (we will clarify this concept in § 4).

While we typically use human-sensible names for entities, this is not necessary for automated systems (although it simplifies debugging!). We have learned (from rather bitter experience) to ruthlessly ban any code that parses an entity-name to extract any meaning; instead, the attributes defined for an entity-kind should encode *anything* that could be extracted from a name. (§11.4 discusses why using names in entity-IDs might not have been the best decision.)

3.2 Allowed relationships

The complete MALT schema consists of a set of entity-kinds (with attributes), a set of relationship-kinds, and a set of allowed relationships. For example, we allow a packet-switch to contain a port, but not vice-versa. These rules constrain producers, but this is good, because it means that model-consuming code need not handle arbitrary relationships.

Relationships can be directed or bidirectional, and 1:1, 1:many, many:1, or (rarely) many:many. We currently allow about 700 relationships between pairs of entity-kinds; this is a small subset of the millions that could be constructed, but we only allow those that support sensible abstractions (a simple form of static validation).

3.3 Multiple levels of abstraction

While MALT's primitive entity-kinds, including those listed above but also others, are sufficient to describe a wide variety of networks, one of the motivations for MALT was that it should allow us to represent multiple levels of abstraction and the relationships between them. Some use cases, for example, involve refining highly-abstracted designs into more concrete ones, but we also may need to reverse an abstraction, and ask (for example) "what abstract thing does this concrete thing belong to?".

We typically create abstraction via hierarchical groupings, such as entity-kinds for:

- **logical switches**: sets of primitive switches, interconnected (e.g., as a "superblock" in a Jupiter [20] network) so that they provide the illusion of one larger switch with more ports than any single switch.
- **trunk links**: parallel sets of links that provide more bandwidth than a single physical link.
- **control domains**: sets of elements controlled by one replica-group of SDN controllers.
- **Geographic elements**: a hierarchy of, e.g., cities, buildings, and spaces within buildings.
- **Dependencies**: sets of entities that could fail together (due to SPOFs, or to sharing a power source) or that must be kept diverse, to avoid accidental SPOFs.

For a WAN, the layering can be quite deep, starting with highly-abstracted city-to-city links, through several levels of trunking to individual L2 links, and through four or five levels of optical-transport-network hierarchy [22, fig. 12].

We also have relationship-kinds that help with abstraction:

- `RK_CONTAINS` for hierarchical containment.
- `RK_AGGREGATES` to indicate, for example, which singleton links are aggregated into a trunk link, or which packet switches are aggregated into a logical switch.
- `RK_TRAVERSES`: e.g., an end-to-end MAC (L2) link is constructed from the ordered *traversal* of a series of L1 links connected by splices and patch panels.

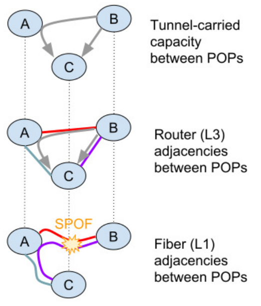

A
B
C
Tunnel-carried capacity between POPs
A
B
C
Router (L3) adjacencies between POPs
SPOF
A
B
C
Fiber (L1) adjacencies between POPs

Figure 4: Layered abstractions in WAN planning

Fig. 4 shows how we can use multiple layers in WAN planning. The top layer shows two WAN tunnels (as in B4 [10]) between POPs B and C, including one via A; the middle layer shows how these tunnels map onto L3 router adjacencies; the bottom layer shows how the L1 fiber path for the $B-C$ L3 adjacency runs through POP A. This means that the $A-B$ fiber path has become a single point of failure (SPOF) Because MALT includes all of these abstractions (abstract flows, IP adjacencies, fiber paths) in the same model, with explicit relationships between them, we can easily map this SPOF back to the tunnels it affects.

3.4 Machine- and human-readable formats

Since MALT is designed to support automation, we normally represent models in a binary format compatible with RPC interfaces. However, developers occasionally need to view (and less often, edit) models or individual components, so we can convert between binary and a textual format. We also have a "shorthand" format for concise text-based representation; this is especially useful for creating test cases.

3.5 Profiles

While we have just one "global" MALT schema, which provides uniformity across all of our networks, we have found it useful to introduce *profiles*, which restrict how the schema is used. We use profiles for purposes including:

- To specialize the schema for certain network types. For example, the profile for a data-center network might assert that the uplinks from aggregation switches are always connected to the downlinks from spine switches, while the profile for an office-campus network might assert that there is always a firewall between the public Internet and the campus network.
- To support evolution, via profile versioning. As we discuss in §10, we are continually evolving both the global

schema and our profiles, which sometimes means changing the way we represent a concept; we need to ensure that model-producers and model-consumers agree on which representation is in use.
- To decouple the release cycles of complex graphs of software systems, so that model producers can move forward without forcing *all* model consumers to migrate at the same time.

A profile is, in effect, a contract between a producer and a consumer. We defined a machine-readable profile specification, which allows us to mechanically check that a model actually meets one more or profiles (as asserted, by the producer, in the model's metadata).

A profile is identified by its name (e.g., "Jupiter"), a major version number, and a minor version number. If we make an incompatible change, such as inserting a new structural layer that turns a one-hop relationship path into a two-hop path, we need to increment the major version.

If two profile-IDs differ only in their minor-version numbers, this implies backward-compatibility: a model with the higher minor-version can be read safely by a consumer already tested against the lower version. The converse does not apply; models with an out-of-date minor version might be missing information that recent consumers expect. (§10 discusses how hard it has been to define "backwards-compatible" in the face of certain coding practices.)

Machine-checkable profiles can express constraints narrower than the entire schema; for example, we can require that certain relationships are present (or not present), or that certain attributes have values within a constrained range. However, our current profile-specification language is not expressive enough to represent certain global policies, such as "no IP address may be assigned to two different interfaces" — we validate that using other means.

4 Division into multiple shards

One might imagine a single model for Google's entire collection of networks, but we actually divide this data into *thousands* of MALT model "shards," for many reasons:
- **Separation of ownership**: Many teams contribute to the design and operation of our networks; things are much simpler when we shard models, so that each shard has a single owner. Such sharding clarifies responsibility, and can avoid the need for complex consistency-maintenance protocols.
- **Distinct profiles**: Different parts of our overall network conform to different profiles (§3.5); sharding allows us to cleanly associate a profile with the data that it covers.
- **Profile evolution**: We cannot change all software instantaneously when introducing a new profile version; instead, we support old versions for a phase-out period. This means that we must represent the same information using multiple profiles, stored as per-version shards. (§10 covers evolution in detail.)
- **Performance**: A single model of our entire network would be too large to fit in the memory of a single server. Also, while many applications extract small sub-models from storage via query RPCs, some do need to retrieve larger subsets; using a "Get" RPC to retrieve one or a few shards is a lot more efficient. (However, if we use too many shards, that leads to per-shard overhead; we try to strike a good balance.)
- **Protection and fault domains**: All software has bugs, and we must defend against security breaches; sharding allows us to limit the damage from a faulty program, and it allows us to set ACLs that restrict users to the shards they actually need. (For example, someone operating on a edge router in Paris does not need access to a datacenter switch in Rome.)
- **Lifecycle stages**: We use several sets of shards to represent distinct points in the lifecycle of a network: e.g., planning, deployment, and operation.
- **Alternative universes**: We need to represent not just a single timeline, but alternative future designs for our networks – e.g., to analyze multiple options for purchasing WAN connectivity or multiple orderings for capacity augments. We especially need to create isolated universes for testing software and operational procedures.

Model sharding requires some support from the query mechanism; §6.2 describes our "multi-shard query" (MSQ) API. It also sometimes requires the same entity to appear in multiple shards (so that no shard has dangling relationships); to avoid violating our uniqueness rule for entity-IDs, we add "linkage" metadata to these repeated entities. Linkage allows us to unambiguously resolve these repeated entities.

Note that, as discussed in §5, each update to a shard creates a new "instance" or version. This is another dimension in which we have many shards.

Given our heavy use of sharding and instances, we need a way to specify a version-consistent snapshot that spans multiple shards; §5.2 describes the "model set" abstraction that supports this.

5 MALT storage

While one might consider treating MALT as a database schema, we instead choose to think of a set of MALT entities and relationships (i.e., a shard) as a named value. One can think of a MALT shard as a file; in fact, we can store a shard as a file, either in binary or in a human-readable format.

We *prefer* to store shards in a purpose-built repository, MALTshop, that provides several important features:
- It is logically centralized, with a single UNIX-like namespace for shards, so we know where to look for any shard (past, present, or future). MALTshop maintains statistics and logs, making it easy to discover who is using the shards and what features are in use.

- It has a distributed, highly-available implementation.
- It provides a basic Get/Put/Patch API, but most model-reading code employs its Query API (see §6).
- Shards are versioned; each update (via a whole-shard Put or diff-based Patch API) creates a new, immutable *instance*, which is permanently bound to a sequence number (but can be mutably bound to a named *label*). Small updates are efficiently implemented via copy-on-write, so the cost of creating and maintaining many versions of a large shard can be relatively low.
- The repository supports ACLs on shards, which provides the basis for security mechanisms. We have not found a need to bear the burden of ACLs on individual entities; those could be layered above MALTshop in an application-specific service, if necessary.

5.1 MALTshop implementation details

MALTshop stores its state in Spanner, a reliable, consistent, distributed SQL database [6], which handles many of the harder problems. The SQL schema has tables for shards, instances, entities, and relationships; an entity's attributes are stored as a SQL blob. Updates to the MALT schema do not require changes to the SQL schema.

While our initial SQL schema supported a simple implementation of MALTshop, as usage increased we realized that the schema did not always support good performance, and read-modify-write operations made it tricky to avoid corruption when a server failed. We are now migrating to a new SQL schema that should improve performance and data integrity; the details are too complex to describe in this paper. Because the SQL schema is *entirely* hidden from all applications, this migration is transparent to all users.

MALTshop uses several kinds of cache, including a client-side cache library that currently supports only "Get" operations, but ultimately should reduce many RPC round-trips, and a server-side cache that greatly reduces the cost of computing diffs between two recent instances of a shard (an operation some of our applications use heavily).

MALTshop, due to Spanner's scalability, itself scales well. We currently store thousands of shards, each with many versions; the largest shards have millions of entities and millions of relationships. Occasionally MALTshop serves thousands of queries per second, but usually the load is lower.

5.2 Model sets

Because we shard our models extensively, and each shard may have many instances (versions), operations that span multiple shards need a way to specify a consistent snapshot, which we support with a "model set" abstraction. When a model-generator creates a set of shard instances, it also obtains a new "model set ID" (MSID), and uses a metadata service to register a binding between the MSID, some attributes, and its constituent shard instances. We therefore usually pass an MSID between systems, rather than lists of instance IDs. The metadata service allows applications to find the latest MSID, or to look up an MSID based on various attributes.

5.3 Discussion: dataflow rather than database

Given our ability to efficiently store (and thus share) immutable versions of MALT shards, it is convenient to think of a single shard instance as a value – a snapshot of a database, rather than a mutable instance of a database. While these values can be quite large, and in many cases an application is only interested in a small subset of a shard, this approach allows us to construct many network management pipelines as dataflow graphs, where streams of MALT shard instances flow between stateless functional operators.

We use these dataflow graphs primarily for planning, design, and analysis applications – systems that operate the existing network do use MALT imperatively (see §11.2). Also, once the planning process must trigger actions with expensive effects (e.g., ordering or installing hardware), we must use imperative operations – isolated to separate shards.

Why not just represent a network topology as a relational database, as is done in many other systems (for example, COOLAID [5])? Some of our previous systems were indeed implemented as RDBMSs, with SQL queries. In our experience, an RDBMS worked nicely for simple use cases, but:

- an RDBMS by itself does not provide clear patterns for new types of abstractions or their graph-type relationships (aggregation, hierarchy, etc.). When we used an RDBMS, we effectively imposed an implicit entity-relationship schema; why not just make that explicit?
- a first-class abstraction of individual shards makes it much simpler to express per-shard profiles, versioning, labels, access controls, and retention policies. It also make it easier to reason about how these shards flow from one system to another – for what-if analysis and isolated software testing, or to parallelize datacenter-expansion projects (as in Fig. 2).
- layering our MALT schema over an SQL schema makes it simpler to do "soft deprecation" of entities, relationships, and attributes, without having to impose those changes on older shard instances or their users.
- layering MALT over SQL makes it easy for us to change the SQL schema, for better performance, without requiring any client changes.
- we can also provide the MALTshop API (albeit with limited performance) on top of a simple file system, which is useful for disaster-recovery scenarios when Spanner might be down or unreachable.

None of these are impossible in SQL (manifestly, since MALTshop expresses all of these with an underlying SQL schema), but by hiding the SQL schema from clients via MALT's abstractions, we make our clients far less brittle. Overall, we have found the dataflow approach far easier to reason about, and in some cases, more efficient. In a few real-time applications (e.g., SDN controllers), we do express a network's topology in a private, purpose-built, in-memory database (with update-in-place).

Other graph-processing systems likewise have avoided RDBMSs, for reasons such as articulated by Hunger *et al.* [9] and for Pregel [16].

6 Querying MALT models

While our model-producing software tends to generate an entire shard in one operation, our model-consuming software generally does not operate on all entities in a shard, but rather only on small subsets. In this respect, our software differs from traditional graph-processing algorithms [16]. (There are exceptions: systems for drain-impact analysis, or WAN cost optimization, do look at entire networks.)

Model consumers extract subsets that meet some predicate via a *query language*, which walks an entity-relationship graph to extract a chosen subset model. Typical queries might be of the form "find all of the top-of-rack switches in datacenter X", "find the switches managed by SDN controller Y", "find all the cable strands that share connector C", or "given port P1 on switch S1, find the corresponding port P2 on switch S2 such that P1 is connected to P2."

Designing a query language for MALT has been surprisingly hard; we are on our second language (and we also toyed with the idea of using a Datalog-style language). Our first query language was sufficient, but it was often hard for users to understand how to express a correct and efficient query. Our second language is easier to use, because it operates on paths through the E-R graph, rather than sets of entities; paths are the more natural abstraction.

Sometimes it is difficult or impossible to use a single query to return exactly the right subset; this leads to a pattern where the application issues a remote query to retrieve a larger-than-desired sub-model, and then applies a local (in-memory) query to further refine the results. In some cases, it is simpler, or even necessary, to post-process the query results using a traditional programming language.

Sometimes people ask "if that's so hard, why not just use SQL?" MALTshop effectively compiles MALT queries to SQL queries, so on the one hand: sure! but on the other hand, these SQL queries are substantially more complex, and also deeply depend on the underlying SQL schema, which we do not want to expose to applications (because we have already had to revise it several times.) We also want to use the same language for both MALTshop and efficiently querying in-memory shards, as shown in Fig. 5; SQL would not support that.

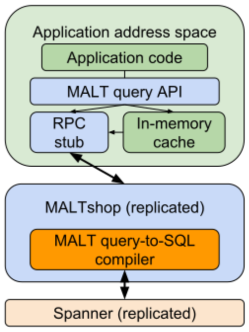


Figure 5: Query layering

In our current language, a query is expressed as a sequence of commands. Each command operates on a "frontier" of active "nodes," which (approximately) are references to entities. Query commands can move the nodes around the input model[3] along relationship edges. As each active node moves around the input model, the query execution engine keeps track of the path it took.

The output is one or more result models, optionally annotated with labels, which includes all active nodes at the end of the query, plus the full path they took (relationships and stepped-over entities) to get to their final position. Some commands remove ("prune") active nodes; these also generally remove, from the result, earlier entities that the node previously visited, if not also visited by another path.

Queries always start with a `find` command, which looks at all entities in the input model, subject to constraints such as an entity-kind and/or some attribute values; e.g.:

```
find EK_PACKET_SWITCH/tor17
```

to start the query at a packet switch named "tor17", or

```
find EK_VLAN { id: 13 }
```

to start the query at all VLANs with a VLAN-ID of 13.

Queries often involve following relationships, e.g.:

```
find EK_PACKET_SWITCH/tor17
  until RK_CONTAINS EK_PORT
```

to return all of the ports contained in that switch.

The language includes many other commands; we lack space to describe the full language.

Query implementation: Queries traverse relationships in a model, marking paths to keep or prune. For complex queries, this may require many iterations through the model.

Queries can be executed locally in-memory, or remotely by MALTshop. For locally-indexed models, these iterations are inexpensive. However, MALTshop has to translate queries into repeated calls to its SQL database. To make this efficient, the query engine requests SQL data in batches.

6.1 "Canned queries"

Queries that are simple to state in English may turn into long sequences of commands. These have proved challenging to write, for many of our users. They can also be fragile with respect to profile changes; when complex, profile-dependent queries are scattered across code owned by many developers, profile change inevitably leads to bugs. Therefore, we have a library of "canned queries." When a profile owner creates a new profile version, that engineer is also responsible for creating (and testing!) a new version of any canned query affected by that change. This gives the responsibility for managing complex queries to the experts on the underlying representations.

[3]That is, change the binding between an active node and an entity.

We define canned queries as needed. For example, one canned query might return all of the L2 links between a given pair of switches; another might return the rack where a given line card is located (useful when trying to repair that card).

6.2 Multi-shard queries

As described in §4, we split the representation of our entire network into many sets of shards. However, some applications would like to form queries that span shard boundaries. Also, we want the freedom to revise our sharding plan (we have done this several times), and we do not want model-consuming code to depend on that plan. Therefore, most applications that query MALTshop actually use a *multi-shard query* (MSQ) API, which allows a query to specify a set of shard pathnames (using wildcards), rather than a single shard; MALTshop then executes the query against a view composed of those shards.

MSQ is efficient, because MaltShop's underlying SQL schema has an index that identifies entities appearing in multiple shards, and limits the queries to only those shards. Performing MSQ in-memory is not feasible, due to the size and time it takes to load and index all the shards.

The introduction of MSQ significantly simplified many applications. For example, prior to MSQ, code looking for "peer ports" at the boundaries between datacenter and WAN networks had to issue separate queries in multiple shards, using the output of the first query to compose the second one. Also, the code had to know which specific shards to query. Code using MSQ does this in one query, and knows much less about shard boundaries.

7 Software infrastructure

In addition to MALTshop (§5) we have developed a library to provide common functions for MALT developers, and additional software to help us use and manage MALT models.

7.1 Model-generation systems

We do not want humans to create detailed models via direct editing; this would be tedious and unreliable. Instead, humans generate highly-abstracted representations (typically via GUIs or text editing) that become input to model-generation software that produces concrete models. We sometimes do this in multiple steps, where the MALT output from one step becomes the input to future steps. At each step, the models become more detailed, based on late-bound design decisions and/or more-specialized profile-specific code.

We have been migrating to a "declarative dataflow" approach to model generation (as in Figs. 1 and 2), away from early systems that used imperative inputs ("add this switch") and that packaged all model-generation steps into one execution. Imperative, non-modular systems (not surprisingly) turned out to be hard to maintain, evolve, and test.

At each step in such a dataflow graph, we can apply our automatic profile-checker (§3.5) to detect some software bugs or incomplete inputs.

7.2 Model-visualization systems

While MALT is designed to support automation, humans often need to look at models. We have visualization systems to support two distinct use cases:

- **Network visualization**: Network operators, capacity planners, and customers want to visualize their network topologies, without knowing how these are represented in MALT. Our network visualizer GUI displays network nodes and the links between them, with statistics and other attributes, and lets a user zoom between abstraction levels. MALT's inherent support for multiple levels of abstraction made this tool easier to write.[4]
- **Model visualization**: Developers of MALT software and models want to visualize the structure of their models, rather than of the network. Our MALTviewer allows them to navigate through entity-relationship graphs, with integral support for the MALT query language.

We considered developing a GUI-based tool to create and edit MALT models, but so far, creating models by expansion of concise, high-level intent (as in §7.1) has sufficed.

8 What does not belong in MALT?

While MALT is central to our network management systems, we do not believe it should be extended beyond expressing topology. People have sought to add other kinds of network-management data (sometimes just to exploit MALTshop rather than investing in another storage system), but these typically do not fit well into an entity-relationship schema. Other categories deserve more-appropriate representations, including:

- **Generated configuration** for devices and SDN controllers; our config generators read MALT models, but their output data belongs in OpenConfig YANG models [18]. We built a "ConfigStore" service more suited to this use case than MALTshop is, because access patterns (especially queries) to "config" are quite different.
- **Policies** for how to use the network topology, such as BGP policies. We believe these are most accessible to the operators who manage these policies when they are expressed using a configuration representation, such as OpenConfig or a vendor-specific one. We allowed some early users of MALT to embed BGP policies in the schema, but that proved to be awkward and complex. (Alternatively, Propane [2] is a domain-specific language for expressing BGP policy.)

[4]A similar tool allows us to visualize the network overlaid onto a geographic map; this is especially useful for planning WAN and campus links that must avoid single points of failure. However, this tool still gets its data from a predecessor to MALT.

- **Abstracted forwarding tables** (AFTs) that represent the observed FIBs in our network; these are useful for applications such as Traffic Engineering and our Bandwidth Enforcer system [13], and for techniques such as Header Space Analysis [11]. AFTs are similar to OpenFlow [17] rules, and might be suitable for representing ACLs, although today we use a DSL for ACLs.
- **Allocated resources**, such as IP addresses, Autonomous System Numbers (ASNs), and ports on switches and patch panels. Since we must not allocate any one of these resources to multiple "owners," these are best represented in a database that supports modify-in-place operations, exactly what we would rather not do for declarative topology models.
- **Inventory**, including support for SOX compliance and other finance-related applications. Often these records exist before we have a topology to place them in.
- **Monitoring data** from SNMP and similar systems. MALT is not efficient at representing time-series data, and Google already has robust, massively-scalable systems for monitoring and alerting, so while we do distill some data from monitoring pipelines to correlate it with topology, as in Fig. 1, we do not use MALT as the primary representation for this data.

We tie these representations together using foreign keys, including MALT entity-IDs (e.g., an AFT is tied to a particular MALT "control point").

It can be tricky to define a bright line between "topology" (appropriate to represent in MALT) and non-topology data. Partly this is driven by a need to store some information in two places, for efficiency and availability – for example, we allocate blocks of IP addresses from an IP-address DB, and then record these blocks in MALT models as attributes of subnetwork entities. However, when we debate "does this data belong in MALT?" the usual reason for the complexity of the debate is that we had not quite got our taxonomy right; relatively few cases have been truly hard calls.

9 Schema design principles and processes

The MALT schema must allow us to represent a broad variety of network designs completely and consistently; code with special cases for different structures is likely to be unreliable and hard to maintain. Schema design turned out to be harder than we expected; we have learned several principles that were not obvious to us at the start. We could not create good abstractions *a priori* for a complex, evolving set of networks, but had to test and refine our abstractions against our experience with many real-life use cases[5].

One meta-lesson was that we needed to establish a formal MALT Review Board (MRB), composed of experienced schema designers, who could take a company-wide and long-term view of proposed changes. Prior to establishing the MRB, the schema accreted many ad hoc, inconsistent, or duplicative features. Using a multi-person board, rather than a single owner, to review schema changes also allows us to parallelize the work, and to maintain consistency as employees come and go. We also have a written "style guide," both as advice to engineers proposing schema changes, and to guide new MRB members. However, our weekly MRB meeting constantly finds new issues to debate at length.

9.1 Orthogonality

We value uniformity: we want our tools to process models for many different kinds (and generations) of networks, without lots of special cases. Sometimes this is straightforward; the MRB often prevents proposals attempting to create a new entity-kind (EK) for an existing concept, or a narrower-than-necessary proposal for a new concept.

We also value simplicity; initially we thought this meant that we should be conservative in creating EKs. However, we learned that overloading one entity-kind with concepts that are not similar enough leads to the use of subtypes (expressed as attributes), which creates complexity in model-consuming code and in the formulation of queries.

We developed two tests to define "similar enough?":

- Do the various use cases share the same relationship structure, or would one expect different relationships based on the subtype? If the latter, we prefer to use a distinct ("orthogonal") EK for each use case, rather than having a kitchen sink of relationships for an EK.
- Do the use cases mostly share the same entity attributes, or are there several mostly-disjoint subsets of the attributes, based on subtype? If the latter, we prefer multiple EKs. (Subtyping violates the "prefer composition over inheritance" principle.)

These are not rigid rules. Sometimes we must guess about future use cases, or just make an arbitrary decision.

An example of why we need these rules: we initially defined `EK_PORT` to refer to singleton ports, trunk ports, VLAN ports, and patch-panel ports. This "simple" structure leads to models where a VLAN `EK_PORT` can contain a trunk `EK_PORT` which can contain multiple singleton `EK_PORTs` – and queries on ports have a lot of complexity to distinguish which subtype they care about. We ended up with over 60 possible relationships involving `EK_PORT`, and about the same number of attributes, most of which are never used together (which makes it hard to check whether a model producer has properly populated the attributes and relationships required for specific use cases).[6]

Entity attributes often use enumerated types. We learned to value multiple, orthogonal attributes over the superficially-

[5] And as we learned these lessons and have evolved our schema, we have also learned just how hard evolution itself can be; see §10.

[6] We have only partially fixed this mess, because lots of existing code uses `EK_PORT`, but without explicitly indicating for which use case.

simpler goal of "fewer attributes." For example, initially "Vendor ID" was an implicit proxy for "switch stack operating system." We had to break that out as a separate "OS" attribute, rather than creating enum values representing the cross-product of vendor-ID and several other features.

9.2 Separation of aspects

We initially modeled a router as a single EK_DEVICE entity. Since routers have lots of substructures, we used lots of attributes to define their various aspects. However, we now model these distinct aspects as explicit entities, separating data-plane aspects, control-plane aspects, and physical ("sheet-metal-plane") aspects. So, for example, we model a simple packet switch with this relationship structure:

```
EK_CHASSIS RK_CONTAINS EK_PACKET_SWITCH
EK_CHASSIS RK_CONTAINS EK_CONTROL_POINT
EK_CONTROL_POINT RK_CONTROLS EK_PACKET_SWITCH
```

This allows model consumers (and their queries) to focus on specific subgraphs: for example, systems that analyze network performance or utilization focus on the data plane subgraph (EK_PACKET_SWITCH, EK_INTERFACE, EK_*_LINK, etc.), while systems involved in hardware installation focus on the physical subgraph (EK_RACK, EK_CHASSIS, etc.) Ultimately, this allows most systems to work correctly no matter how we re-organize control-plane, data-plane, and physical-plane structures. (Especially in space-limited testbeds, we often need to use non-standard packaging, which required lot of special cases in software before we separated these aspects.)

Somewhat similarly, we also use separate entity-kinds to represent the abstract intent and the concrete realization of a complex structure, such as a Clos network. In our model-generation pipeline, we use the "intent" entities in the inputs to a step that generates the concrete entities; the output includes the inputs, tied to the concrete entities via RK_REALIZED_BY relationships, so that the intent is visible to consumers of the concrete models.

10 Profile evolution

We must continually change our MALT schema, both to handle new kinds of network designs, and to rethink how we have represented things in the past (taxonomy is hard; we have made a *lot* of mistakes). Additions are fairly easy, but other changes create a lot of pain for software maintainers, and the risk of misinterpretation.

While we expected the schema evolution, the challenges that created were much larger than we initially expected. Because many systems interact via models, and models persist (often for years), we have had to create processes and software tools to ensure compatibility. Also, networking concepts can evolve faster within one company than in the public Internet – and faster than our ability to rapidly upgrade our software base, or educate software engineers.

We developed several mechanisms to cope with evolution:

- **Stability rules**, to avoid schema changes that create unnecessary churn (but these can lead to accretion of the equivalent of "dead code" in the schema).
- **Profiles and profile versions**, as discussed in §3.5. Before we had profiles, evolution was especially painful, because there was no explicit way for a consumer to know which of several possible interpretations to place on a model.

  Because profiles are versioned, our model-generators can simultaneously produce shards for the same network design in several versions; this allows us to update producers and consumers independently.
- **Feature flags**, which specify explicitly which individual features are enabled (or disabled) in models produced for a given profile version, so that consumer code can condition its behavior on the presence or absence of specific features, rather than complex logic based on version numbers. For example, a feature-flag might indicate that a given profile version supports IPv6 address prefixes; these might have been previously allowed in the MALT *schema*, but not generated in the *models* until a given profile version.
- **Profile-version deprecation policies**, which allow us to (gently) force consumers to migrate off of old versions, so the producers do not have to support them forever.
- **Canned queries**, described in §6.1, which insulate the less-complex model consumers from profile-specific details. (Not all consumers can fully rely on canned queries for insulation, and model producers might have to be migrated for each profile.)

As mentioned in §3.5, if two profiles differ only in their minor-version number, code for the lower version should be able to consume models of the higher version. Unfortunately, it has been tricky to define rules for "backwards compatibility," especially in the face of some fragile coding practices. For example, code often does a switch statement on an enumeration attribute. Not all code properly handles a newly-defined enum value; some code crashes, and other code blithely treats the new value as equivalent to an older value, leading to silent failures. We have thus gradually become more cautious about allowing profile changes without incrementing the major version, but such increments often lead to tedious "requalification" of software.

These mechanisms also make it easier to change our shard boundaries: canned queries hide the boundaries from most users; for others, we use a profile-version change to signal a sharding change.

Overall, these techniques are helpful, but not sufficient, to avoid the pain of profile evolution; we continually look for new approaches.

Other systems have had to grapple with evolution, with varying success. For example, the QUIC protocol designers made version-negotiation a fundamental aspect of the protocol design, and then relied on it to rapidly iterate through many versions [14].

11 Lessons and challenges

After using and evolving MALT for several years, we have come to appreciate both the benefits of this approach, and some of its challenges.

11.1 Benefits and challenges of adoption

Prior to MALT, we had many non-interoperable topology representations, including multiple formal ones, many spreadsheets, drawings, and sometimes just folklore. Convergence on a single, well-curated, machine-focused representation has yielded benefits including far simpler interoperability between systems, dramatic reduction in some code complexity (and complete elimination of a few code bases), and a motivation to invest in improved data quality.

However, converting from older representations, especially the few that were already the basis for islands of automation, has been painful. (Parts of our network-management world that had no prior formal representation have often been easier to migrate to MALT, because they have little existing code to worry about.)

Things that make representation conversion difficult include differences in model structure (e.g., tabular vs. entity-relationship); differences in the handling of defaults; differences in rules for constructing names; and (quite frequently) incomplete or inaccurate documentation on the semantics of the old representation. We also discovered that, since older representations tended to be weakly validated (at best), and existing consumer code was tolerant of missing or inaccurate data, we kept running into data we could not understand (or wrongly thought we did).

We learned, as MALT supported an increase in data-driven automation, that a good representation cannot save us from dirty data. If the data is missing or wrong, automation fails in depressing ways.

We observed an interesting pattern: operators typically start by focusing on operating the "built" system, and see no need for formal representation of the entire lifecycle (as in Figs. 1 and 2). Then, as the network gets larger and more complex, they gradually realize they must be involved in planning and design phases, so that the network is actually operable and well-documented. This pattern reinforces the value of a uniform, multi-abstraction-layer representation.

Similarly, network test engineers initially believe they can manage their small, idiosyncratic testbeds informally. But they learn that end-to-end network deployment and management processes in a testbed needs to work exactly as they do in production (or else you have not actually tested everything), which motivates formal modeling even for testbeds. A side benefit is that they waste less time doing manual work for which automated, MALT-based tooling exists. (However, testbeds are often weird, which adds complexity to model-generation tools.)

11.2 Designing via a declarative approach

Our prior network-design systems were mostly imperative, with complex APIs of the form "add these links" or "remove this switch." Humans naturally think imperatively, but these APIs became an impediment to modular composition, due to their complexity. They also made it difficult to create isolated universes for what-if analysis and testing.

MALT supports (but does not require) a declarative design process, in which each stage process tells the next stage what it wants, not how to get it. APIs become simpler; all conceptual complexity is now explicit in the models. To create a testbed or what-if model, we can simply create a copy of an existing model, modify it, and then run the normal pipeline.

Our network-design world is not fully declarative: since humans still create the top-level design intent, our UIs support some imperative operations. In cases where operators want to make minor, rapid changes, we primarily use MALT imperatively.

11.3 The dangers of string parsing

We are trying to stamp out string-parsing, because parsers (and regular expressions) embody assumptions about how strings encode information. When we later must change a format (e.g., to allow longer fields), we have found it hard to search code for all of the relevant parsers (coders have many creative ways to parse strings), so we discover many of these only when something breaks at run-time. Instead, we have learned to discourage string-encoded data, and to provide explicit attributes and/or relationship. (E.g., if an entity is named "router.paris.33" then it needs a relationship to the entity for "paris" and an `index_in_parent` attribute.)

11.4 Human-readable names vs. UUIDs

§3.1 describes how MALT entity-IDs are (entity-kind, entity-name) tuples. This eased the initial adoption of MALT, because our existing code and operator procedures all used human-sensible names. In hindsight, we should have used opaque universally unique identifiers (UUIDs) as primary keys, and kept the display name as an attribute. With name-based entity-IDs, renaming becomes hard, because we have to track down all references to an entity-name, including those in external systems.

Name-based IDs can be especially tricky when creating designs for abstract components, which need IDs, before we know their ultimate names, which are often late-bound. We currently ameliorate that problem by a "placeholder" mechanism that lets us rebind references to entities, but names held in non-MALT systems still lead to problems. UUIDs introduce their own challenges, which we lack space to describe.

12 Related work

Many large enterprises face similar network-management challenges, so prior papers have described systems related to MALT, but almost none have focused specifically on the challenges of a multi-layer abstraction and how to evolve it. Propane/AT [3] does describe ways to model abstract groups of switches and their adjacencies, but not how to include finer levels of detail. Most other work uses the term "topology" only to refer to IP or routing-session adjacencies.

COOLAID [5] used a declarative-language approach based on Datalog. COOLAID focused on reasoning about configuration management of an existing network, rather than topology design or abstract intent. (Their emphasis on reasoning is complementary to MALT.)

Facebook's Robotron [21] followed a "configuration-as-code" paradigm, rather than a declarative representation; this appears to limit its use to a single administrative domain, and to complicate its use for what-if analyses. Robotron does not handle multi-step or concurrent design changes [21, §8]. Robotron supports "high-level design intent"; it is unclear if this extends to abstractions for capacity planning.

Alibaba's NetCraft [15] manages "the life cycle of network configuration, including the generation, update, transition and diagnosis of [configuration]," using a multi-layer graph, but apparently not abstractions that support network planning or design. One layer represents a BGP mesh (contrasted to Propane [2] which represents BGP *policy*).

Google's Zero-Touch Networking (ZTN) [12] provides an automation framework which utilizes both MALT and OpenConfig [18] as interoperable data representations.

MALT's design was inspired by UML [19] and the DMTF "Common Information Model" standard, which uses an object-relationship representation of "managed elements." UML focuses mostly on modeling systems, not on network topologies. DTMF includes a layer-3 interface profile [7], but also seems to have little coverage of network topology *per se*. (OpenConfig might well subsume this aspect of DTMF.)

SmartFrog [8] was a framework for automated management of configuration for multi-component software systems. While it differs from MALT in numerous ways, SmartFrog's *templates* are similar to MALT's entity-kinds, and SmartFrog also addressed lifecycle issues.

13 Conclusions and future work

Our experience has shown that, while it was challenging to design both MALT's representation and an ecosystem that supports its widespread use, we have gained great value from a declarative, multi-layered approach to representing network topology. MALT supports the full lifecycle of our networks, allowing us to make knowledge explicit in our models, rather than hidden in code.

Others wishing to learn from our experience might want to consider these challenges that surprised us:

- **Shared schemas need curation**: a representation whose goal is to create interoperability between disparate teams and processes cannot be evolved by uncoordinated accretion; curation by a centralized team (the MRB § 9) has a hard but necessary job.
- **Support for evolution**: we added explicit support for schema evolution, and explicit profiles, later than we should have (§ 10).
- **Simplicity $\neq$ fewer concepts**: Our initial attempts to limit the number of concepts (entity-kinds) led to complexity via overloading; in retrospect, orthogonality via many simple concepts brings simplicity (§ 9.1).
- **Query language**: Our three-layered approach (canned queries at the top, a powerful query language in the middle, and a well-hidden SQL layer at the bottom) seems to create the right balance between expressive power vs. ease of use, but we struggled to find that balance (§ 6).
- **Migration**: Migrating users from previous representations created considerable pain, some of which could have been avoided if we had learned our other lessons much sooner.

We also had some positive surprises:

- **Support for lots of models**: all of our previous systems maintained just one model or database. MALTshop's ability to give near-arbitrary names to models, and to use immutable versions, enabled many use cases we had not initially considered, and enabled sharding (§ 5).
- **Extension to other domains**: Because our software base (§7) is largely agnostic to the MALT schema itself, it could support multiple domains with distinct schemas. We plan to explore modeling network-like domains, such as liquid cooling and power distribution.

Future work: We believe MALT can be extended to cover several other kinds of networks; this remains future work. These include cloud networks, and especially "hybrid" networks that include both cloud and "on-premises" enterprise networks. (MALTshop would need to support multiple, isolated namespaces for shards and for entities.) MALT could also be extended to explicitly model Software-as-a-Service connectivity.

MALT might support wireless networks, with some redesign. The features that make wireless networks "interesting," such as mobile nodes, hidden terminals, and dynamic channel fading, will challenge some implicit assumptions we made for MALT.

Acknowledgements

Hundreds (at least) of our colleagues have contributed code, models, queries, documents, and sometimes painful experience to the MALT journey, and continue to do so; we cannot possibly name them all. We also thank the NSDI reviewers, and our shepherd Sujata Banerjee, for their helpful feedback.

References

[1] Ajay Kumar Bangla, Alireza Ghaffarkhah, Ben Preskill, Bikash Koley, Christoph Albrecht, Emilie Danna, Joe Jiang, and Xiaoxue Zhao. Capacity planning for the Google backbone network. In *International Symposium on Mathematical Programming (ISMP)*, 2015.

[2] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proc. SIGCOMM*, pages 328–341, 2016.

[3] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network Configuration Synthesis with Abstract Topologies. In *Proc. PLDI*, page 437–451, 2017.

[4] Peter Pin-Shan Chen. The Entity-relationship Model—Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

[5] Xu Chen, Yun Mao, Z. Morley Mao, and Jacobus Van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *Proc. CoNEXT*, 2010.

[6] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[7] John Parchem (ed.). Network Management Layer3 Interface Profile. Technical report, Distributed Management Task Force, Inc., 2018.

[8] Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The SmartFrog Configuration Management Framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, January 2009.

[9] Michael Hunger, Ryan Boyd, and William Lyon. RDBMS & Graphs: SQL vs. Cypher Query Languages. https://neo4j.com/blog/sql-vs-cypher-query-languages/, 2016.

[10] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. In *Proc. SIGCOMM*, pages 3–14, 2013.

[11] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. NSDI*, 2012.

[12] Bikash Koley. The Zero Touch Network. In *Proc. Intl. Conf. on Network and Service Management*, 2016. https://ai.google/research/pubs/pub45687.

[13] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proc. SIGCOMM '15*, 2015.

[14] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proc. SIGCOMM*, pages 183–196, 2017.

[15] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. Automatic Life Cycle Management of Network Configurations. In *Proc. Workshop on Self-Driving Networks (SelfDN)*, pages 29–35, 2018.

[16] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proc. SIGMOD*, pages 135–146, 2010.

[17] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[18] OpenConfig Working Group. Data models and APIs. http://www.openconfig.net/projects/models/.

[19] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[20] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim

Wanderer, Urs Hoelzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. SIGCOMM*, 2015.

[21] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proc. SIGCOMM*, pages 426–439, 2016.

[22] Timothy P. Walker. Optical Transport Network (OTN) Tutorial. https://www.itu.int/ITU-T/studygroups/com15/otn/OTNtutorial.pdf.

[23] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks. In *Proc. NSDI*, pages 221–234, Boston, MA, February 2019.

A Example of MALT modeling

We use the Jupiter datacenter fabric design [20] to illustrate MALT modeling with a fine-grained, although incomplete and very simplified, example. In general, we use one shard (§ 4) for each such fabric.

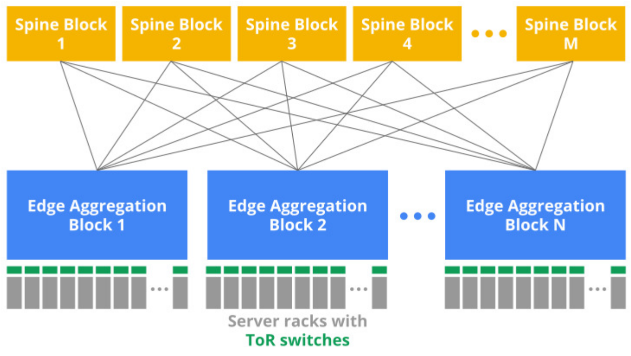


Figure 6: Overall structure of Jupiter (after [20])

Figs. 6–8 provide a summary of Jupiter's components: a fabric is made up of *aggregation blocks* connected to each other via a set of *spine blocks*. An aggregation block is a set of *middle blocks*, each of which is a Clos fabric made up of many switch chips. (Each spine block is also one Clos fabric.) Aggregation blocks connect to ToR switches. (A special kind of aggregation block, a *border router*, has the same structure but connects to WAN links rather than to ToR uplinks.)

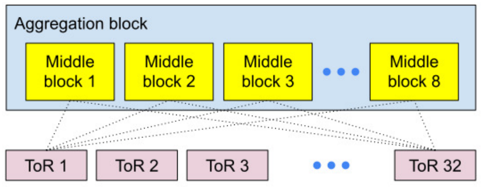


Figure 7: Jupiter aggregation block (after [20])

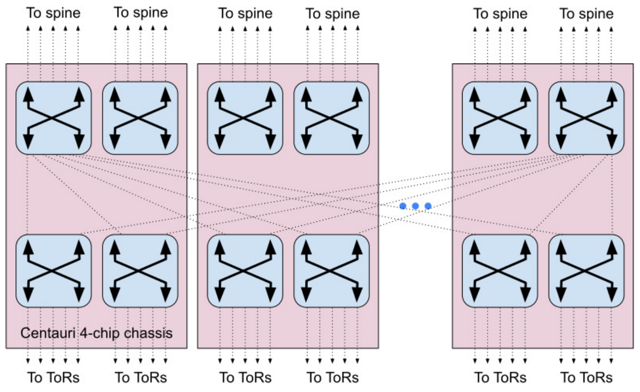


Figure 8: Jupiter middle block (after [20])

As described in [20] and shown in Fig. 8, we package four switch chips on one "Centauri" *chassis*. A ToR is one such chassis; a middle block is four chassis; we package multiple chassis into each *rack*.

Thus, a Jupiter has several hierarchies: a "data plane" hierarchy of packet switch chips and larger switch-like abstractions; a "sheet metal plane" hierarchy of racks, chassis, and chips; and a "control plane" hierarchy.

We can model the top-level data-plane hierarchy for a small Jupiter called "ju1" as (using our shorthand syntax):

```
EK_JUPITER/ju1 RK_CONTAINS EK_AGG_BLOCK/ju1.a1
EK_JUPITER/ju1 RK_CONTAINS EK_AGG_BLOCK/ju1.a2
EK_JUPITER/ju1 RK_CONTAINS EK_AGG_BLOCK/ju1.a3
EK_JUPITER/ju1 RK_CONTAINS EK_AGG_BLOCK/ju1.a4

EK_JUPITER/ju1 RK_CONTAINS EK_SPINE_BLOCK/ju1.s1
 ...
EK_JUPITER/ju1 RK_CONTAINS EK_SPINE_BLOCK/ju1.s4

EK_AGG_BLOCK/ju1.a1 RK_CONTAINS EK_TOR/ju1.a1.t1
...
EK_AGG_BLOCK/ju1.a1 RK_CONTAINS EK_TOR/ju1.a1.t32
```

An aggregation block abstractly contains multiple middle blocks:

```
EK_AGG_BLOCK/ju1.a1 RK_CONTAINS
  EK_MIDDLE_BLOCK/ju1.a1.m1
...
EK_AGG_BLOCK/ju1.a1 RK_CONTAINS
    EK_MIDDLE_BLOCK/ju1.a1.m8
```

A middle block is a "logical switch" abstractly containing multiple switch chips:

```
EK_MIDDLE_BLOCK/ju1.a1.m1 RK_CONTAINS
  EK_PACKET_SWITCH/ju1.a1.m1.c1
...
EK_MIDDLE_BLOCK/ju1.a1 RK_CONTAINS
    EK_PACKET_SWITCH/ju1.a1.m1.c16
```

Chips within middle blocks are connected, so we also have to indicate the 8 *ports* on each switch chip:

```
  EK_PACKET_SWITCH/ju1.a1.m1.c16 RK_CONTAINS
    EK_PORT/ju1.a1.m1.c16.p1
  ...
  EK_PACKET_SWITCH/ju1.a1.m1.c16 RK_CONTAINS
    EK_PORT/ju1.a1.m1.c16.p16
```

and then the 64 bidirectional L3 links between the ports in the upper and lower layers of the block – just *one* of which would become (note that links in MALT are unidirectional):

```
  EK_PORT/ju1.a1.m1.c1.p9 RK_ORIGINATES
    EK_LOGICAL_PACKET_LINK/ju1.a1.m1.l1f
  EK_PORT/ju1.a1.m1.c9.p1 RK_TERMINATES
    EK_LOGICAL_PACKET_LINK/ju1.a1.m1.l1f
  EK_PORT/ju1.a1.m1.c9.p1 RK_ORIGINATES
    EK_LOGICAL_PACKET_LINK/ju1.a1.m1.l1r
  EK_PORT/ju1.a1.m1.c1.p9 RK_TERMINATES
    EK_LOGICAL_PACKET_LINK/ju1.a1.m1.l1r
```

Fig. 3 shows how L3 links traverse L2 links, which in turn might traverse multiple fibers, patch panels, etc. (not shown).

We similarly need to represent the internal connections in each spine block (just as in a middle block) and the connections between middle-block ports and spine-block ports, and those between middle-block ports and ToR ports. (Note that middle blocks in an aggregation block are not directly connected to each other.)

There are clearly a *lot* of links in a Jupiter network. Therefore, we use automated tools to design the cables that bundle these links. Those tools need to know the spatial locations of the ports to which these links connect; therefore, MALT also allows us to represent the physical containment hierarchy: each rack contains 16 chassis, which each contains 4 packet switches, which each contains 16 ports.

```
EK_RACK/ju1.a1.m1 RK_CONTAINS EK_CHASSIS/ju1.a1.m1.chass1
...
EK_RACK/ju1.a1.m1 RK_CONTAINS EK_CHASSIS/ju1.a1.m1.chass16

EK_CHASSIS/ju1.a1.m1.chass1 RK_CONTAINS
  EK_PACKET_SWITCH/ju1.a1.m1.c1
...
EK_CHASSIS/ju1.a1.m1.chass1 RK_CONTAINS
  EK_PACKET_SWITCH/ju1.a1.m1.c4

EK_PACKET_SWITCH/ju1.a1.m1.c16 RK_CONTAINS
   EK_PORT/ju1.a1.m1.c16.p1
... (as above)
```

Jupiter is a software-defined network, so we also represent the network control plane and its connectivity – information required to automatically generate configuration for packet switches and for controllers. We start by abstracting the switch-local CPU as a "control point" for the switch – what SDN controllers will communicate with:

```
EK_CHASSIS/ju1.a1.m1.chass1 RK_CONTAINS
  EK_CONTROL_POINT/ju1.a1.m1.chass1
EK_CONTROL_POINT/ju1.a1.m1.chass1 RK_CONTAINS
  EK_INTERFACE/ju1.a1.m1.chass1.if1
EK_INTERFACE/ju1.a1.m1.chass1.if1 RK_TRAVERSES
  EK_PORT/ju1.a1.m1.chass1.port1
EK_PORT/ju1.a1.m1.chass1.port1 RK_ORIGINATES
  EK_LOGICAL_PACKET_LINK/...
```

Note that MALT allows us to give two different entities the same name, as long as they have different entity-kinds – here, the control point has the same name as its containing chassis.

Since a Centauri chassis has one CPU for 4 switch chips, we model this as:

```
EK_CONTROL_POINT/ju1.a1.m1.chass1 RK_CONTROLS
  EK_PACKET_SWITCH/ju1.a1.m1.c1
...
EK_CONTROL_POINT/ju1.a1.m1.chass1 RK_CONTROLS
  EK_PACKET_SWITCH/ju1.a1.m1.c4
```

We can then represent the relationships between a set of SDN switches and their controllers via indirection: all switches with the same set of controller replicas are grouped into a "control domain" (we generally have one control domain per aggregation block, to provide fault tolerance):

```
EK_CONTROL_DOMAIN/ju1.dom1 RK_CONTAINS
  EK_CONTROL_POINT/ju1.a1.m1.chass1
...
EK_CONTROL_DOMAIN/ju1.dom16 RK_CONTAINS
  EK_CONTROL_POINT/ju1.a16.m8.chass1
```

This indirection allows us to represent a pool of controller replicas responsible for all switches in a control domain:

```
EK_CONTROLLER/ju1.controller1.1 RK_CONTROLS
  EK_CONTROL_DOMAIN/ju1.dom1
...
EK_CONTROLLER/ju1.controller1.4 RK_CONTROLS
  EK_CONTROL_DOMAIN/ju1.dom1
```

We can also represent – omitted here due to lack of space – that controllers run on a pool of dedicated server machines, how these machines are arranged in racks, are connected to the network, etc.

Attributes: So far, this appendix has only described entities and relationships. Each entity has a set of attributes; space only permits us to show a few (simplified) examples. A specific port might include these attributes:

```
port_attr: <
  device_port_name: "port-1/24"
  openflow: <
    of_port_number: 24
  >
  port_role: PR_SINGLETON
  port_attributes: <
    physical_capacity_bps: 40000000000
  >
>
```

while a specific L3 interface might include these:

```
interface_attr: <
  address: <
    ipv4: <
      address: "10.1.2.3"
      prefixlen: 32
    >
    ipv6: <
      address: "1111:2222:3333:4444::"
      prefixlen: 64
    >
  >
>
```

Firecracker: Lightweight Virtualization for Serverless Applications

Alexandru Agache
Amazon Web Services

Marc Brooker
Amazon Web Services

Andreea Florescu
Amazon Web Services

Alexandra Iordache
Amazon Web Services

Anthony Liguori
Amazon Web Services

Rolf Neugebauer
Amazon Web Services

Phil Piwonka
Amazon Web Services

Diana-Maria Popa
Amazon Web Services

Abstract

Serverless containers and functions are widely used for deploying and managing software in the cloud. Their popularity is due to reduced cost of operations, improved utilization of hardware, and faster scaling than traditional deployment methods. The economics and scale of serverless applications demand that workloads from multiple customers run on the same hardware with minimal overhead, while preserving strong security and performance isolation. The traditional view is that there is a choice between virtualization with strong security and high overhead, and container technologies with weaker security and minimal overhead. This tradeoff is unacceptable to public infrastructure providers, who need both strong security and minimal overhead. To meet this need, we developed Firecracker, a new open source Virtual Machine Monitor (VMM) specialized for serverless workloads, but generally useful for containers, functions and other compute workloads within a reasonable set of constraints. We have deployed Firecracker in two publically-available serverless compute services at Amazon Web Services (Lambda and Fargate), where it supports millions of production workloads, and trillions of requests per month. We describe how specializing for serverless informed the design of Firecracker, and what we learned from seamlessly migrating Lambda customers to Firecracker.

1 Introduction

Serverless computing is an increasingly popular model for deploying and managing software and services, both in public cloud environments, e.g., [4, 16, 50, 51], as well as in on-premises environments, e.g., [11,41]. The serverless model is attractive for several reasons, including reduced work in operating servers and managing capacity, automatic scaling, pay-for-use pricing, and integrations with sources of events and streaming data. Containers, most commonly embodied by Docker, have become popular for similar reasons, including reduced operational overhead, and improved manageability. Containers and Serverless offer a distinct economic advantage over traditional server provisioning processes: multitenancy allows servers to be shared across a large number of workloads, and the ability to provision new functions and containers in milliseconds allows capacity to be switched between workloads quickly as demand changes. Serverless is also attracting the attention of the research community [21,26,27,44,47], including work on scaling out video encoding [13], linear algebra [20, 53] and parallel compilation [12].

Multitenancy, despite its economic opportunities, presents significant challenges in isolating workloads from one another. Workloads must be isolated both for security (so one workload cannot access, or infer, data belonging to another workload), and for operational concerns (so the noisy neighbor effect of one workload cannot cause other workloads to run more slowly). Cloud instance providers (such as AWS EC2) face similar challenges, and have solved them using hypervisor-based virtualization (such as with QEMU/KVM [7, 29] or Xen [5]), or by avoiding multi-tenancy and offering bare-metal instances. Serverless and container models allow many more workloads to be run on a single machine than traditional instance models, which amplifies the economic advantages of multi-tenancy, but also multiplies any overhead required for isolation.

Typical container deployments on Linux, such as those using Docker and LXC, address this density challenge by relying on isolation mechanisms built into the Linux kernel. These mechanisms include *control groups* (*cgroups*), which provide process grouping, resource throttling and accounting; *namespaces*, which separate Linux kernel resources such as process IDs (PIDs) into namespaces; and *seccomp-bpf*, which controls access to syscalls. Together, these tools provide a powerful toolkit for isolating containers, but their reliance on a single operating system kernel means that there is a fundamental tradeoff between security and code compatibility. Container implementors can choose to improve security by limiting syscalls, at the cost of breaking code which requires the restricted calls. This introduces difficult tradeoffs: implementors of serverless and container services can choose between hypervisor-based virtualization (and the potentially unacceptable overhead related to it), and Linux containers (and the related compatibility vs. security tradeoffs). We built Firecracker because we didn't want to choose.

Other projects, such as Kata Containers [14], Intel's Clear Containers, and NEC's LightVM [38] have started from a similar place, recognizing the need for improved isolation, and choosing hypervisor-based virtualization as the way to achieve that. QEMU/KVM has been the base for the majority of these projects (such as Kata Containers), but others (such as LightVM) have been based on slimming down Xen. While QEMU has been a successful base for these projects, it is a large project ($>$ 1.4 million LOC as of QEMU 4.2), and has focused on flexibility and feature completeness rather than overhead, security, or fast startup.

With Firecracker, we chose to keep KVM, but entirely replace QEMU to build a new Virtual Machine Monitor (VMM), device model, and API for managing and configuring MicroVMs. Firecracker, along with KVM, provides a new foundation for implementing isolation between containers and functions. With the provided minimal Linux guest kernel configuration, it offers memory overhead of less than 5MB per container, boots to application code in less than 125ms, and allows creation of up to 150 MicroVMs per second per host. We released Firecracker as open source software in December 2018[1], under the Apache 2 license. Firecracker has been used in production in Lambda since 2018, where it powers millions of workloads and trillions of requests per month.

Section 2 explores the choice of an isolation solution for Lambda and Fargate, comparing containers, language VM isolation, and virtualization. Section 3 presents the design of Firecracker. Section 4 places it in context in Lambda, explaining how it is integrated, and the role it plays in the performance and economics of that service. Section 5 compares Firecracker to alternative technologies on performance, density and overhead.

1.1 Specialization

Firecracker was built specifically for serverless and container applications. While it is broadly useful, and we are excited to see Firecracker be adopted in other areas, the performance, density, and isolation goals of Firecracker were set by its intended use for serverless and containers. Developing a VMM for a clear set of goals, and where we could make assumptions about the properties and requirements of guests, was significantly easier than developing one suitable for all uses. These simplifying assumptions are reflected in Firecracker's design and implementation. This paper describes Firecracker in context, as used in AWS Lambda, to illustrate why we made the decisions we did, and where we diverged from existing VMM designs. The specifics of how Firecracker is used in Lambda are covered in Section 4.1.

[1] https://firecracker-microvm.github.io/

Firecracker is probably most notable for what it does not offer, especially compared to QEMU. It does not offer a BIOS, cannot boot arbitrary kernels, does not emulate legacy devices nor PCI, and does not support VM migration. Firecracker could not boot Microsoft Windows without significant changes to Firecracker. Firecracker's process-per-VM model also means that it doesn't offer VM orchestration, packaging, management or other features — it replaces QEMU, rather than Docker or Kubernetes, in the container stack. Simplicity and minimalism were explicit goals in our development process. Higher-level features like orchestration and metadata management are provided by existing open source solutions like Kubernetes, Docker and `containerd`, or by our proprietary implementations inside AWS services. Lower-level features, such as additional devices (USB, PCI, sound, video, etc), BIOS, and CPU instruction emulation are simply not implemented because they are not needed by typical serverless container and function workloads.

2 Choosing an Isolation Solution

When we first built AWS Lambda, we chose to use Linux containers to isolate functions, and virtualization to isolate between customer accounts. In other words, multiple functions for the same customer would run inside a single VM, but workloads for different customers always run in different VMs. We were unsatisfied with this approach for several reasons, including the necessity of trading off between security and compatibility that containers represent, and the difficulties of efficiently packing workloads onto fixed-size VMs. When choosing a replacement, we were looking for something that provided strong security against a broad range of attacks (including microarchitectural side-channel attacks), the ability to run at high densities with little overhead or waste, and compatibility with a broad range of unmodified software (Lambda functions are allowed to contain arbitrary Linux binaries, and a significant portion do). In response to these challenges, we evaluated various options for re-designing Lambda's isolation model, identifying the properties of our ideal solution:

Isolation: It must be safe for multiple functions to run on the same hardware, protected against privilege escalation, information disclosure, covert channels, and other risks.

Overhead and Density: It must be possible to run thousands of functions on a single machine, with minimal waste.

Performance: Functions must perform similarly to running natively. Performance must also be consistent, and isolated from the behavior of neighbors on the same hardware.

Compatibility: Lambda allows functions to contain arbitrary Linux binaries and libraries. These must be supported without code changes or recompilation.

Fast Switching: It must be possible to start new functions and clean up old functions quickly.

Soft Allocation: It must be possible to over commit CPU, memory and other resources, with each function consuming only the resources it needs, not the resources it is entitled to.

Some of these qualities can be converted into quantitative goals, while others (like isolation) remain stubbornly qualitative. Modern commodity servers contain up to 1TB of RAM, while Lambda functions use as little as 128MB, requiring up to 8000 functions on a server to fill the RAM (or more due to soft allocation). We think of overhead as a percentage, based on the size of the function, and initially targeted 10% on RAM and CPU. For a 1024MB function, this means 102MB of memory overhead. Performance is somewhat complex, as it is measured against the function's entitlement. In Lambda, CPU, network, and storage throughput is allocated to functions proportionally to their configured memory limit. Within these limits, functions should perform similarly to bare metal on raw CPU, IO throughput, IO latency and other metrics.

2.1 Evaluating the Isolation Options

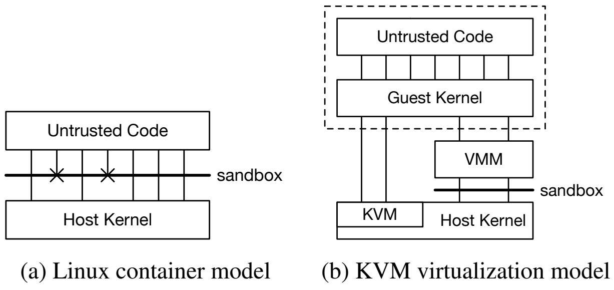
Broadly, the options for isolating workloads on Linux can be broken into three categories: *containers*, in which all workloads share a kernel and some combination of kernel mechanisms are used to isolate them; *virtualization*, in which workloads run in their own VMs under a hypervisor; and *language VM isolation*, in which the language VM is responsible for either isolating workloads from each other or from the operating system. [2]

Figure 1 compares the security approaches between Linux containers and virtualization. In Linux containers, untrusted code calls the host kernel directly, possibly with the kernel surface area restricted (such as with seccomp-bpf). It also interacts directly with other services provided by the host kernel, like filesystems and the page cache. In virtualization, untrusted code is generally allowed full access to a guest kernel, allowing all kernel features to be used, but explicitly treating the guest kernel as untrusted. Hardware virtualization and the VMM limit the guest kernel's access to the privileged domain and host kernel.

2.1.1 Linux Containers

Containers on Linux combine multiple Linux kernel features to offer operational and security isolation. These features include: *cgroups*, providing CPU, memory and other resource

[2] It's somewhat confusing that in common usage *containers* is both used to describe the mechanism for packaging code, and the typical implementation of that mechanism. Containers (the abstraction) can be provided without depending on containers (the implementation). In this paper, we use the term *Linux containers* to describe the implementation, while being aware that other operating systems provide similar functionality.



(a) Linux container model (b) KVM virtualization model

Figure 1: The security model of Linux containers (a) depends directly on the kernel's sandboxing capabilities, while KVM-style virtualization (b) relies on security of the VMM, possibly augmented sandboxing

limits; *namespaces*, providing namespacing for kernel resources like user IDs (uids), process IDs (pids) and network interfaces; *seccomp-bpf*, providing the ability to limit which syscalls a process can use, and which arguments can be passed to these syscalls; and *chroot*, proving an isolated filesystem. Different Linux container implementations use these tools in different combinations, but *seccomp-bpf* provides the most important security isolation boundary. The fact that containers rely on syscall limitations for their security represents a tradeoff between security and compatibility. A trivial Linux application requires 15 unique syscalls. Tsai et al [57] found that a typical Ubuntu Linux 15.04 installation requires 224 syscalls and 52 unique *ioctl* calls to run without problems, along with the */proc* and */sys* interfaces of the kernel. Nevertheless, meaningful reduction of the kernel surface is possible, especially as it is reasonable to believe that the Linux kernel has more bugs in syscalls which are less heavily used [32].

One approach to this challenge is to provide some of the operating system functionality in userspace, requiring a significantly smaller amount of kernel functionality to provide the programmer with the appearance of a fully featured environment. Graphene [56], Drawbridge [45], Bascule [6] and Google's gvisor [15] take this approach. In environments running untrusted code, container isolation is not only concerned with preventing privilege escalation, but also in preventing information disclosure side channels (such as [19]), and preventing communication between functions over covert channels. The richness of interfaces like */proc* have showed this to be challenging (CVE-2018-17972 and CVE-2017-18344 are recent examples).

2.1.2 Language-Specific Isolation

A second widely-used method for isolating workloads is leveraging features of the language virtual machine, such as the Java Virtual Machine (JVM) or V8. Some language VMs (such as V8's *isolates* and the JVM's security managers) aim to run multiple workloads within a single process, an approach which introduces significant tradeoffs between functionality (and compatibility) and resistance to side channel attacks such as Spectre [30, 39]. Other approaches, such as Chromium site isolation [46], Alto [31] and SAND [1] use a process per trust domain, and instead aim to prevent the code from escaping from the process or accessing information from beyond the process boundary. Language-specific isolation techniques were not suitable for Lambda or Fargate, given our need to support arbitrary binaries.

2.1.3 Virtualization

Modern virtualization uses hardware features (such as Intel VT-x) to provide each sandbox an isolated environment with its own virtual hardware, page tables, and operating system kernel. Two key, and related, challenges of virtualization are density and overhead. The VMM and kernel associated with each guest consumes some amount of CPU and memory before it does useful work, limiting density. Another challenge is startup time, with typical VM startup times in the range of seconds. These challenges are particularly important in the Lambda environment, where functions are small (so relative overhead is larger), and workloads are constantly changing. One way to address startup time is to boot something smaller than a full OS kernel, such as a unikernel. Unikernels are already being investigated for use with containers, for example in LightVM [38] and Solo5 [59]. Our requirement for running unmodified code targeting Linux meant that we could not apply this approach.

The third challenge in virtualization is the implementation itself: hypervisors and virtual machine monitors (VMMs), and therefore the required *trusted computing base* (TCB), can be large and complex, with a significant attack surface. This complexity comes from the fact that VMMs still need to either provide some OS functionality themselves (type 1) or depend on the host operating system (type 2) for functionality. In the type 2 model, The VMM depends on the host kernel to provide IO, scheduling, and other low-level functionality, potentially exposing the host kernel and side-channels through shared data structures. Williams et al [60] found that virtualization does lead to fewer host kernel functions being called than direct implementation (although more than their libOS-based approach). However, Li et al [32] demonstrate the effectiveness of a 'popular paths' metric, showing that only 3% of kernel bugs are found in frequently-used code paths (which, in our experience, overlap highly with the code paths used by the VMM).

To illustrate this complexity, the popular combination of Linux Kernel Virtual Machine [29] (KVM) and QEMU clearly illustrates the complexity. QEMU contains $>$ 1.4 million lines of code, and can require up to 270 unique syscalls [57] (more than any other package on Ubuntu Linux 15.04). The KVM code in Linux adds another 120,000 lines. The NEMU [24] project aims to cut down QEMU by removing unused features, but appears to be inactive.

Efforts have been made (such as with Muen [9] and Nova [55]) to significantly reduce the size of the Hypervisor and VMM, but none of these minimized solutions offer the platform independence, operational characteristics, or maturity that we needed at AWS.

Firecracker's approach to these problems is to use KVM (for reasons we discuss in section 3), but replace the VMM with a minimal implementation written in a safe language. Minimizing the feature set of the VMM helps reduce surface area, and controls the size of the TCB. Firecracker contains approximately 50k lines of Rust code (96% fewer lines than QEMU), including multiple levels of automated tests, and auto-generated bindings. Intel Cloud Hypervisor [25] takes a similar approach, (and indeed shares much code with Firecracker), while NEMU [24] aims to address these problems by cutting down QEMU.

Despite these challenges, virtualization provides many compelling benefits. From an isolation perspective, the most compelling benefit is that it moves the security-critical interface from the OS boundary to a boundary supported in hardware and comparatively simpler software. It removes the need to trade off between kernel features and security: the guest kernel can supply its full feature set with no change to the threat model. VMMs are much smaller than general-purpose OS kernels, exposing a small number of well-understood abstractions without compromising on software compatibility or requiring software to be modified.

3 The Firecracker VMM

Firecracker is a Virtual Machine Monitor (VMM), which uses the Linux Kernel's KVM virtualization infrastructure to provide minimal virtual machines (MicroVMs), supporting modern Linux hosts, and Linux and OSv guests. Firecracker provides a REST based configuration API; device emulation for disk, networking and serial console; and configurable rate limiting for network and disk throughput and request rate. One Firecracker process runs per MicroVM, providing a simple model for security isolation.

Our other philosophy in implementing Firecracker was to rely on components built into Linux rather than re-implementing our own, where the Linux components offer the right features, performance, and design. For example we pass block IO through to the Linux kernel, depend on Linux's process scheduler and memory manager for handling contention between VMs in CPU and memory, and we use TUN/TAP virtual network interfaces. We chose this path for two reasons. One was implementation cost: high-quality operating system components, such as schedulers, can take decades to get right, especially when they need to deal with multi-tenant workloads on multi-processor machines. The implementations in Linux, while not beyond criticism [36], are well-proven in high-scale deployments.

The other reason was operational knowledge: within AWS, our operators are highly experienced at operating, automating, optimizing Linux systems (for example, Brendan Gregg's books on Linux performance [18] are popular among Amazon teams). Using KVM in Linux, along with the standard Linux programming model, allows our operators to use most of the tools they already know when operating and troubleshooting Firecracker hosts and guests. For example, running *ps* on a Firecracker host will include all the MicroVMs on the host in the process list, and tools like *top*, *vmstat* and even *kill* work as operators expect. While we do not routinely provide access to in-production Firecracker hosts to operators, the ability to use the standard Linux toolset has proven invaluable during the development and testing of our services. In pursuit of this philosophy, Firecracker does sacrifice portability between host operating systems, and inherits a larger trusted compute base.

In implementing Firecracker, we started with Google's Chrome OS Virtual Machine Monitor *crosvm*, re-using some of its components. Consistent with the Firecracker philosophy, the main focus of our adoption of *crosvm* was removing code: Firecracker has fewer than half as many lines of code as *crosvm*. We removed device drivers including USB and GPU, and support for the 9p filesystem protocol. Firecracker and *crosvm* have now diverged substantially. Since diverging from *crosvm* and deleting the unneeded drivers, Firecracker has added over 20k lines of new code, and changed 30k lines. The rust-vmm project[3] maintains a common set of open-source Rust crates (packages) to be shared by Firecracker and crosvm and used as a base by future VMM implementers.

3.1 Device Model

Reflecting its specialization for container and function workloads, Firecracker provides a limited number of emulated devices: network and block devices, serial ports, and partial i8042 (PS/2 keyboard controller) support. For comparison, QEMU is significantly more flexible and more complex, with support for more than 40 emulated devices, including USB, video and audio devices. The serial and i8042 emulation implementations are straightforward: the i8042 driver is less than 50 lines of Rust, and the serial driver around 250. The network and block implementations are more complex, reflecting both higher performance requirements and more inherent complexity. We use *virtio* [40, 48] for network and block devices, an open API for exposing emulated devices from hypervisors. *virtio* is simple, scalable, and offers sufficiently good overhead and performance for our needs. The entire *virtio* block implementation in Firecracker (including MMIO and data structures) is around 1400 lines of Rust.

We chose to support block devices for storage, rather than filesystem passthrough, as a security consideration. Filesystems are large and complex code bases, and providing only block IO to the guest protects a substantial part of the host kernel surface area.

[3]https://github.com/rust-vmm/community

3.2 API

The Firecracker process provides a REST API over a Unix socket, which is used to configure, manage and start and stop MicroVMs. Providing an API allows us to more carefully control the life cycle of MicroVMs. For example, we can start the Firecracker process and pre-configure the MicroVM and only start the MicroVM when needed, reducing startup latency. We chose REST because clients are available for nearly any language ecosystem, it is a familiar model for our targeted developers, and because OpenAPI allows us to provide a machine- and human-readable specification of the API. By contrast, the typical Unix approach of command-line arguments do not allow messages to be passed to the process after it is created, and no popular machine-readable standard exists for specifying structured command-line arguments. Firecracker users can interact with the API using an HTTP client in their language of choice, or from the command line using tools like *curl*.

REST APIs exist for specifying the guest kernel and boot arguments, network configuration, block device configuration, guest machine configuration and cpuid, logging, metrics, rate limiters, and the metadata service. Common defaults are provided for most configurations, so in the simplest use only the guest kernel and one (root) block device need to be configured before the VM is started.

To shut down the MicroVM, it is sufficient to kill the Firecracker process, or issue a *reboot* inside the guest. As with the rest of Firecracker, the REST API is intentionally kept simple and minimal, especially when compared to similar APIs like Xen's Xenstore.

3.3 Rate Limiters, Performance and Machine Configuration

The machine configuration API allows hosts to configure the amount of memory and number of cores exposed to a MicroVM, and set up the *cpuid* bits that the MicroVM sees. While Firecracker offers no emulation of missing CPU functionality, controlling *cpuid* allows hosts to hide some of their capabilities from MicroVMs, such as to make a heterogeneous compute fleet appear homogeneous.

Firecracker's block device and network devices offer built-in rate limiters, also configured via the API. These rate limiters allow limits to be set on operations per second (IOPS for disk, packets per second for network) and on bandwidth for each device attached to each MicroVM. For the network, separate limits can be set on receive and transmit traffic. Limiters are implemented using a simple in-memory token bucket, optionally allowing short-term bursts above the base rate, and a one-time burst to accelerate booting. Having rate limiters be configurable via the API allows us to vary limits based on configured resources (like the memory configured for a Lambda function), or dynamically based on demand. Rate limiters play two roles in our systems: ensuring that our storage devices and networks have sufficient bandwidth available for control plane operations, and preventing a small number of busy MicroVMs on a server from affecting the performance of other MicroVMs.

While Firecracker's rate limiters and machine configuration provide the flexibility that we need, they are significantly less flexible and powerful than Linux *cgroups* which offer additional features including CPU credit scheduling, core affinity, scheduler control, traffic prioritization, performance events and accounting. This is consistent with our philosophy. We implemented performance limits in Firecracker where there was a compelling reason: enforcing rate limits in device emulation allows us to strongly control the amount of VMM and host kernel CPU time that a guest can consume, and we do not trust the guest to implement its own limits. Where we did not have a compelling reason to add the functionality to Firecracker, we use the capabilities of the host OS.

3.4 Security

Architectural and micro-architectural side-channel attacks have existed for decades. Recently, the publication of Meltdown [34], Spectre [30], Zombieload [49], and related attacks (e.g. [2, 37, 54, 58]) has generated a flurry of interest in this area, and prompted the development of new mitigation techniques in operating systems, processors, firmware and microcode. Canella et al [10] and ARM [33] provide good summaries of the current state of research. With existing CPU capabilities, no single layer can mitigate all these attacks, so mitigations need to be built into multiple layers of the system. For Firecracker, we provide clear guidance on current side-channel mitigation best-practices for deployments of Firecracker in production[4]. Mitigations include disabling Symmetric MultiThreading (SMT, aka HyperThreading), checking that the host Linux kernel has mitigations enabled (including Kernel Page-Table Isolation, Indirect Branch Prediction Barriers, Indirect Branch Restricted Speculation and cache flush mitigations against L1 Terminal Fault), enabling kernel options like Speculative Store Bypass mitigations, disabling swap and samepage merging, avoiding sharing files (to mitigate timing attacks like Flush+Reload [61] and Prime+Probe [42]), and even hardware recommendations to mitigate RowHammer [28, 43]. While we believe that all of these practices are necessary in a public cloud environment, and enable them in our Lambda and Fargate deployments of Firecracker, we also recognize that tradeoffs exist between performance and security, and that Firecracker consumers in less-demanding environments may choose not to implement some of these mitigations. As with all security mitigations, this is not an end-point, but an ongoing process of understanding and responding to new threats as they surface.

Other side-channel attacks, such as power and temperature, are not addressed by Firecracker, and instead must be handled elsewhere in the system architecture. We have paid careful attention to mitigating these attacks in our own services, but anybody who adopts Firecracker must understand them and have plans to mitigate them.

3.4.1 Jailer

Firecracker's jailer implements an additional level of protection against unwanted VMM behavior (such as a hypothetical bug that allows the guest to inject code into the VMM). The jailer implements a wrapper around Firecracker which places it into a restrictive sandbox before it boots the guest, including running it in a *chroot*, isolating it in *pid* and network namespaces, dropping privileges, and setting a restrictive *seccomp-bpf* profile. The sandbox's *chroot* contains only the Firecracker binary, */dev/net/tun*, cgroups control files, and any resources the particular MicroVM needs access to (such as its storage image). The *seccomp-bpf* profile whitelists 24 syscalls, each with additional argument filtering, and 30 ioctls (of which 22 are required by KVM ioctl-based API).

4 Firecracker In Production

4.1 Inside AWS Lambda

Lambda [51] is a compute service which runs functions in response to events. Lambda offers a number of built-in language runtimes (including Python, Java, NodeJS, and C#) which allows functions to be provided as snippets of code implementing a language-specific runtime interface. A "Hello, World!" Lambda function can be implemented in as few as three lines of Python or Javascript. It also supports an HTTP/REST runtime API, allowing programs which implement this API to be developed in any language, and provided either as binaries or a bundle alongside their language implementation. Lambda functions run within a sandbox, which provides a minimal Linux userland and some common libraries and utilities. When Lambda functions are created, they are configured with a memory limit, and a maximum runtime to handle each individual event[5]. Events include those explicitly created by calling the Lambda *Invoke* API, from HTTP requests via AWS's Application Load Balancer and API Gateway, and from integrations with other AWS services including storage (S3), queue (SQS), streaming data (Kinesis) and database (DynamoDB) services.

Typical use-cases for AWS Lambda include backends for IoT, mobile and web applications; request-response and event-

[4]https://github.com/firecracker-microvm/firecracker/blob/master/docs/prod-host-setup.md

[5]As of early 2019, Lambda limits memory to less than 3GB and runtime to 15 minutes, but we expect these limits to increase considerably over time.

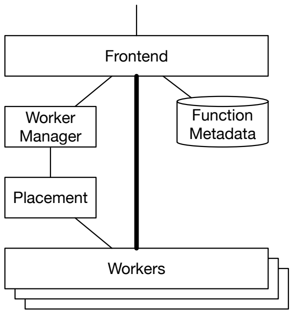


Figure 2: High-level architecture of AWS Lambda event path, showing control path (light lines) and data path (heavy lines)

sourced microservices; real-time streaming data processing; on-demand file processing; and infrastructure automation. AWS markets Lambda as *serverless* compute, emphasizing that Lambda functions minimize operational and capacity planning work, and entirely eliminate per-server operations for most use-cases. Most typical deployments of Lambda functions use them with other services in the AWS suite: S3, SQS, DynamoDB and Elasticache are common companions. Lambda is a large-scale multi-tenant service, serving trillions of events per month for hundreds of thousands of customers.

4.1.1 High-Level Architecture

Figure 2 presents a simplified view of the architecture of Lambda. Invoke traffic arrives at the frontend via the *Invoke* REST API, where requests are authenticated and checked for authorization, and function metadata is loaded. The frontend is a scale-out shared-nothing fleet, with any frontend able to handle traffic for any function. The execution of the customer code happens on the Lambda worker fleet, but to improve cache locality, enable connection re-use and amortize the costs of moving and loading customer code, events for a single function are sticky-routed to as few workers as possible. This sticky routing is the job of the Worker Manager, a custom high-volume (millions of requests per second) low-latency (<10ms 99.9th percentile latency) stateful router. The Worker Manager replicates sticky routing information for a single function (or small group of functions) between a small number of hosts across diverse physical infrastructure, to ensure high availability. Once the Worker Manager has identified which worker to run the code on, it advises the invoke service which sends the payload directly to the worker to reduce round-trips. The Worker Manager and workers also implement a concurrency control protocol which resolves the race conditions created by large numbers of independent invoke services operating against a shared pool of workers.

Each Lambda worker offers a number of *slots*, with each slot providing a pre-loaded execution environment for a function. Slots are only ever used for a single function, and a single concurrent invocation of that function, but are used for many serial invocations of the function. The MicroVM and the process the function is running in are both re-used, as illustrated by Listing 1 which will return a series of increasing numbers when invoked with a stream of serial events.

Listing 1 Lambda function illustrating slot re-use. The returned number will count up over many invokes.

```
var i = 0;
exports.handler = async (event, context) => {
    return i++;
};
```

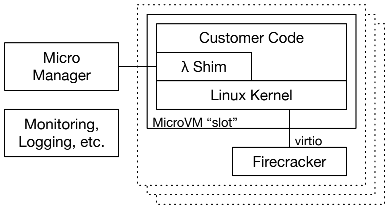


Figure 3: Architecture of the Lambda worker

Where a slot is available for a function, the Worker Manager can simply perform its lightweight concurrency control protocol, and tell the frontend that the slot is available for use. Where no slot is available, either because none exists or because traffic to a function has increased to require additional slots, the Worker Manager calls the Placement service to request that a new slot is created for the function. The Placement service in turn optimizes the placement of slots for a single function across the worker fleet, ensuring that the utilization of resources including CPU, memory, network, and storage is even across the fleet and the potential for correlated resource allocation on each individual worker is minimized. Once this optimization is complete — a task which typically takes less than 20ms — the Placement service contacts a worker to request that it creates a slot for a function. The Placement service uses a time-based lease [17] protocol to lease the resulting slot to the Worker Manager, allowing it to make autonomous decisions for a fixed period of time.

The Placement service remains responsible for slots, including limiting their lifetime (in response to the life cycle of the worker hosts), terminating slots which have become idle or redundant, managing software updates, and other similar activities. Using a lease protocol allows the system to both maintain efficient sticky routing (and hence locality) and have clear ownership of resources. As part of its optimization responsibilities, the placement service also consumes load and health data for each slot in each worker.

4.1.2 Firecracker In The Lambda Worker

Figure 3 shows the architecture of the Lambda worker, where Firecracker provides the critical security boundary required to run a large number of different workloads on a single server.

Each worker runs hundreds or thousands of MicroVMs (each providing a single *slot*), with the number depending on the configured size of each MicroVM, and how much memory, CPU and other resources each VM consumes. Each MicroVM contains a single sandbox for a single customer function, along with a minimized Linux kernel and userland, and a shim control process. The MicroVM is a primary security boundary, with all components assuming that code running inside the MicroVM is untrusted. One Firecracker process is launched per MicroVM, which is responsible for creating and managing the MicroVM, providing device emulation and handling VM exits.

The shim process in each MicroVM communicates through the MicroVM boundary via a TCP/IP socket with the MicroManager, a per-worker process which is responsible for managing the Firecracker processes. MicroManager provides slot management and locking APIs to placement, and an event invoke API to the Frontend. Once the Frontend has been allocated a slot by the WorkerManager, it calls the MicroManager with the details of the slot and request payload, which the MicroManager passes on to the Lambda shim running inside the MicroVM for that slot. On completion, the MicroManager receives the response payload (or error details in case of a failure), and passes these onto the Frontend for response to the customer. Communicating into and out of the MicroVM over TCP/IP costs some efficiency, but is consistent with the design principles behind Firecracker: it keeps the MicroManager process loosely coupled from the MicroVM, and re-uses a capability of the MicroVM (networking) rather than introducing a new device. The MicroManager's protocol with the Lambda shim is important for security, because it is the boundary between the multi-tenant Lambda control plane, and the single-tenant (and single-function) MicroVM. Also, on each worker is a set of processes that provides monitoring, logging, and other services for the worker. Logs and metrics are provided for consumption by both humans and automated alarming and monitoring systems, and metrics are also provided back to Placement to inform its view of the load on the worker.

The MicroManager also keeps a small pool of pre-booted MicroVMs, ready to be used when Placement requests a new slot. While the 125ms boot times offered by Firecracker are fast, they are not fast enough for the scale-up path of Lambda, which is sometimes blocking user requests. Fast booting is a first-order design requirement for Firecracker, both because boot time is a proxy for resources consumed during boot, and because fast boots allow Lambda to keep these spare pools small. The required mean pool size can be calculated with Little's law [35]: the pool size is the product of creation rate and creation latency. Alternatively, at 125ms creation time, one pooled MicroVM is required for every 8 creations per second.

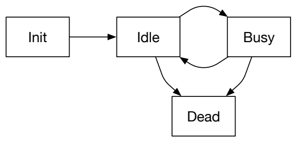


Figure 4: State transitions for a single slots on a Lambda worker

4.2 The Role of Multi-Tenancy

Soft-allocation (the ability for the platform to allocate resources on-demand, rather than at startup) and multi-tenancy (the ability for the platform to run large numbers of unrelated workloads) are critical to the economics of Lambda. Each slot can exist in one of three states: *initializing*, *busy*, and *idle*, and during their lifetimes slots move from *initializing* to *idle*, and then move between *idle* and *busy* as invokes flow to the slot (see Figure 4). Slots use different amounts of resources in each state. When they are *idle* they consume memory, keeping the function state available. When they are *initializing* and *busy*, they use memory but also resources like CPU time, caches, network and memory bandwidth and any other resources in the system. Memory makes up roughly 40% of the capital cost of typical modern server designs, so *idle* slots should cost 40% of the cost of busy slots. Achieving this requires that resources (like CPU) are both soft-allocated and oversubscribed, so can be sold to other slots while one is *idle*.

Oversubscription is fundamentally a statistical bet: the platform must ensure that resources are kept as busy as possible, but some are available to any slot which receives work. We set some compliance goal X (e.g., 99.99%), so that functions are able to get all the resources they need with no contention $X\%$ of the time. Efficiency is then directly proportional to the ratio between the Xth percentile of resource use, and mean resource use. Intuitively, the mean represents revenue, and the Xth percentile represents cost. Multi-tenancy is a powerful tool for reducing this ratio, which naturally drops approximately with $\sqrt{N}$ when running N uncorrelated workloads on a worker. Keeping these workloads uncorrelated requires that they are unrelated: multiple workloads from the same application, and to an extent from the same customer or industry, behave as a single workload for these purposes.

4.3 Experiences Deploying and Operating Firecracker

Starting in 2018, we migrated Lambda customers from our first isolation solution (based on containers per function, and AWS EC2 instances per customer) to Firecracker running on AWS EC2 bare metal instances, with no interruption to availability, latency or other metrics. Each Lambda slot exists for at most 12 hours before it is recycled. Simply changing the recycling logic to switch between Firecracker and legacy implementations allowed workloads to be migrated with no change in behavior.

We took advantage of users of Lambda inside AWS by migrating their workloads first, and carefully monitoring their metrics. Having access to these internal customer's metrics and code reduced the risk of early stages of deployment, because we didn't need to rely on external customers to inform us of subtle issues. This migration was mostly uneventful, but we did find some minor issues. For example, our Firecracker fleet has Symmetric MultiThreading (SMT, aka Hyperthreading) disabled [52] while our legacy fleet had it enabled[6]. Migrating to Firecracker changed the timings of some code, and exposed minor bugs in our own SDK, and in Apache Commons HttpClient [22, 23]. Once we moved all internal AWS workloads, we started to migrate external customers. This migration has been uneventful, despite moving arbitrary code provided by hundreds of thousands of AWS customers.

Throughout the migration, we made sure that the metrics and logs for the Firecracker and legacy stacks could be monitored independently. The team working on the migration carefully compared the metrics between the two stacks, and investigated each case where they were diverged significantly. We keep detailed per-workload metrics, including latency and error-rate, allowing us to use statistical techniques to proactively identify anomalous behavior. Architecturally, we chose to use a common production environment for both stacks as far as possible, isolating the differences between the stacks behind a common API. This allowed the migration team to work independently, shifting the target of the API traffic with no action from teams owning other parts of the architecture.

Our deployment mechanism was also designed to allow fast, safe, and (in some cases) automatic rollback, moving customers back to the legacy stack at the first sign of trouble. Rollback is a key operational safety practice at AWS, allowing us to mitigate customer issues quickly, and then investigate. One case where we used this mechanism was when some customers reported seeing DNS-related performance issues: we rolled them back, and then root-caused the issue to a misconfiguration causing DNS lookups not to be cached inside the MicroVM.

One area of focus for our deployment was building mechanisms to patch software, including Firecracker and the guest and host kernels, quickly and safely and then audit that patches have reached the whole fleet. We use an *immutable infrastructure* approach, where we patch by completely re-imaging the host, implemented by terminating and re-launching our AWS EC2 instances with an updated machine image (AMI). We chose this approach based on a decade of experience operating AWS services, and learning that it is extremely difficult to keep software sets consistent on large-scale mutable fleets (for example, package managers like *rpm* are non-deterministic, producing different results on across a fleet). It also illustrates the value of building higher-level services like Lambda on lower-level services like AWS EC2: we didn't need to build any host management or provisioning infrastructure.

[6]We disable SMT on the Firecracker fleet as a sidechannel mitigation. On the legacy fleet the same threats are mitigated by pinning VMs to cores.

5 Evaluation

In Section 2, we described six criteria for choosing a mechanism for isolation in AWS Lambda: Isolation, Overhead, Performance, Compatibility, Fast Switching and Soft Allocation. In this section we evaluate Firecracker against these criteria as well as other solutions in this space. We use Firecracker v0.20.0 as the base line and use QEMU v4.2.0 (statically compiled with a minimal set of options) for comparison. We also include data for the recently released Intel Cloud Hypervisor [25], a VMM written in Rust sharing significant code with Firecracker, while targeting a different application space.

Our evaluation is performed on an EC2 *m5d.metal* instance, with two Intel Xeon Platinum 8175M processors (for a total of 48 cores with hyper-threading disabled), 384GB of RAM, and four 840GB local NVMe disks. The base OS was Ubuntu 18.04.2 with kernel 4.15.0-1044-aws. The configuration and scripts used for all our experiments, and the resulting data, is publicly available[7].

5.1 Boot Times

Boot times of MicroVMs are important for serverless workloads like Lambda. While Lambda uses a small local pool of MicroVMs to hide boot latency from customers, the costs of switching between workloads (and therefore the cost of creating new MicroVMs) is very important to our economics. In this section we compare the boot times of different VMMs. The boot time is measured as the time between when VMM process is forked and the guest kernel forks its `init` process. For this experiment we use a minimal `init` implementation, which just writes to a pre-configured IO port. We modified all VMMs to call `exit()` when the write to this IO port triggers a VM exit.

All VMMs directly load a Linux 4.14.94 kernel via the command line (i.e. the kernel is directly loaded by the VMM and not some bootloader). The kernel is configured with the settings we recommend for MicroVMs (minimal set of kernel driver and no kernel modules), and we use a file backed minimal root filesystem containing the `init` process. The VMs are configured with a single vCPU and 256MB of memory.

Figure 5 shows the cumulative distribution of (wall-clock) kernel boot times for 500 samples executed serially, so only one boot was taking place on the system at a time. Firecracker results are presented in two ways: end-to-end, including forking the Firecracker process and configuration through the API; and pre-configured where Firecracker has already been set up

[7]`https://github.com/firecracker-microvm/nsdi2020-data`

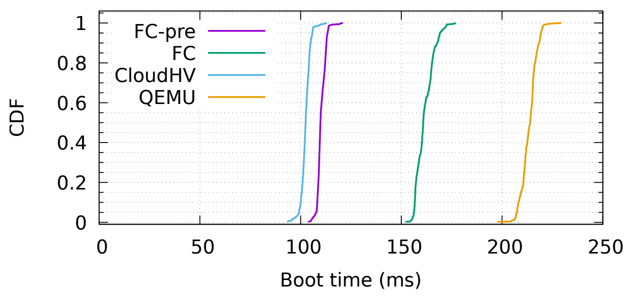


Figure 5: Cumulative distribution of wall-clock times for starting MicroVMs in serial, for pre-configured Firecracker (FC-pre), end-to-end Firecracker, Cloud Hypervisor, and QEMU.

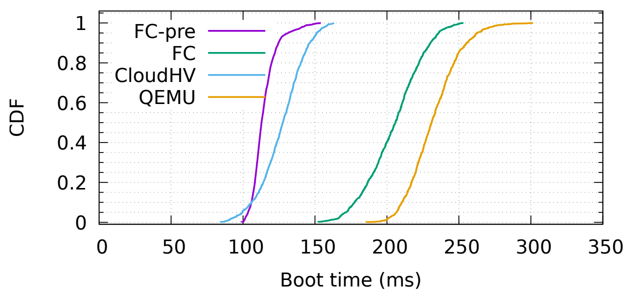


Figure 6: Cumulative distribution of wall-clock times for starting 50 MicroVMs in parallel, for pre-configured Firecracker (FC-pre), end-to-end Firecracker, Cloud Hypervisor,and QEMU.

through the API and time represent the wall clock time from the final API call to start the VM until the `init` process gets executed.

Pre-configured Firecracker and the Cloud Hypervisor perform significantly better than QEMU, both on average and with a tighter distribution, booting twice as fast as QEMU. The end-to-end Firecracker boot times are somewhat in-between, which is expected since we currently perform several API calls to configure the Firecracker VM. Cloud Hypervisor boots marginally faster than pre-configured Firecracker. We suspect subtle differences in the VM emulation to be the reason.

We then booted 1000 MicroVMs with 50 VMs launching concurrently. While any highly-concurrent test will give results that vary between runs, the results in Figure 6 are representative: Firecracker and QEMU perform similarly for end-to-end comparisons, roughly maintaining the 50ms gap already seen with the serial boot times above. Like with serial boot times, pre-configured Firecracker and Cloud Hypervisor show significantly better results, with pre-configured Firecracker out-performing Cloud Hypervisor both in average boot times as well as a tighter distribution with a 99th percentile of 146ms vs 158ms. We also repeated the experiment with starting 100 MicroVMs concurrently and see a similar trend, albeit with a slightly wider distribution with the 99th percentile for pre-configure Firecracker going up to 153ms.

Based on these results, we are investigating a number of optimizations to Firecracker. We could move the Firecracker configuration into a single combined API call would close the gap between the pre-configured and end-to-end boot times. We also investigate to move some of the VM initialization into the VM configuration phase, e.g., allocating the VM memory and loading the kernel image during the configuration API call. This should improve considerably the pre-configured boot times.

The choice of VMM is, of course, only one factor impacting boot times. For example, both Firecracker and Cloud Hypervisor are capable of booting uncompressed Linux kernels while QEMU only boots compressed kernel images. Cloud Hypervisor is also capable of booting compressed kernel. Decompressing the kernel during boot adds around 40ms. The kernel configuration also matters. In the same test setup, the kernel which ships with Ubuntu 18.04 takes an additional 900ms to start! Part of this goes to timeouts when probing for legacy devices not emulated by Firecracker, and part to loading unneeded drivers. In our kernel configuration, we exclude almost all drivers, except the *virtio* and serial drivers that Firecracker supports,build all functionality into the kernel (avoiding modules), and disable any kernel features that typical container and serverless deployments will not need. The compressed kernel built with this configuration is 4.0MB with no modules, compared to the Ubuntu 18.04 kernel at 6.7MB with 44MB of modules. We also recommend a kernel command line for Firecracker which, among other things, disables the standard logging to the serial console (saving up to 70ms of boot time). Note, we use similar optimizations for the other VMMs in iur tests.

For the tests above, all MicroVMs were configured without networking. Adding a statically configured network interface adds around 20ms to the boot times for Firecracker and Cloud Hypervisor and around 35ms for QEMU. Finally, QEMU requires a BIOS to boot. For our test we use *qboot* [8], a minimal BIOS, which reduces boot times by around 20ms compared to the default BIOS. Overall the boot times compare favourable to those reported in the literature, for example [38] reported boot times for an unmodified Linux kernel of 180ms.

5.2 Memory overhead

For Lambda, memory overhead is a key metric, because our goal is to sell all the memory we buy. Lower memory overhead also allows for a higher density of MicroVMs. We measure the memory overhead for the different VMMs as the difference between memory used by the VMM process and the configured MicroVM size. To measure the size of the VMM process we parse the output of the `pmap` command and add up all non-shared memory segments reported. This way our

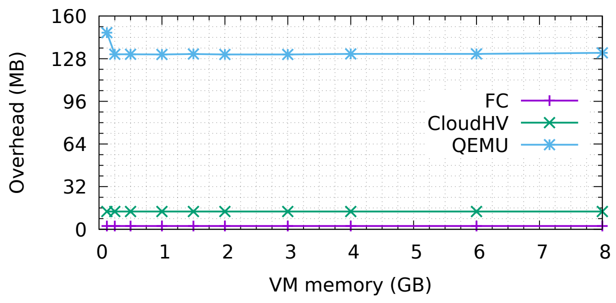


Figure 7: Memory overhead for different VMMs depending on the configured VM size.

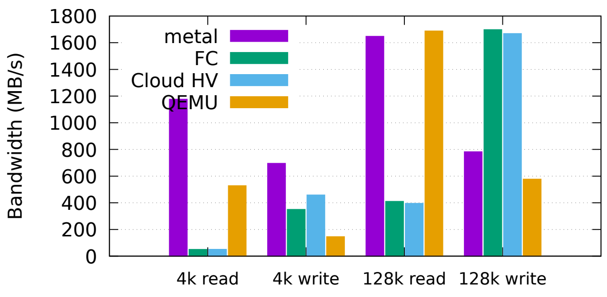


Figure 8: IO throughput on EC2 m5d.metal and running inside various VMs.

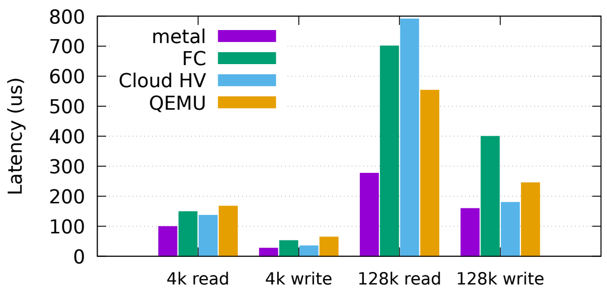


Figure 9: 99th percentile IO latency on EC2 m5d.metal and running inside various VMMs.

calculation excludes the size of the VMM binary itself.

Figure 7 shows that all three VMMs have a constant memory overhead irrespective of the configured VM size (with the exception of QEMU, where for a 128MB sized VM, the overhead is slightly higher). Firecracker has the smallest overhead (around 3MB) of all VMM sizes measured. Cloud Hypervisors overhead is around 13MB per VM. QEMU has the highest overhead of around 131MB per MicroVM. In particular for smaller MicroVMs, QEMU's overhead is significant.

We also measured the memory available inside the MicroVM using the `free` command. The difference in memory available between the different VMMs is consistent with the data presented above.

5.3 IO performance

In this section, we look at the IO performance of the different VMMs both for disks and networking. For all tests we use VMs configured with 2 vCPUs and 512MB of memory. For block IO performance tests we use `fio` [3], and rate limiting was disabled in Firecracker. `fio` is configured to perform random IO directly against the block device, using direct IO through libaio, and all tests were backed by the local NVMe SSDs on the *m5d.metal* server. Figure 8 shows the performance of random read and write IO of small (4kB) and large (128kB) blocks (queue depth 32). The results reflect two limitations in Firecracker's (and Cloud Hypervisor's) current block IO implementation: it does not implement flush-to-disk so high write performance comes at the cost of durability (particularly visible in 128kB write results), and it is currently limited to handling IOs serially (clearly visible in the read results). The hardware is capable of over 340,000 read IOPS (1GB/s at 4kB), but the Firecracker (and Cloud Hypervisor) guest is limited to around 13,000 IOPS (52MB/s at 4kB). We expect to fix both of these limitations in time. QEMU clearly has a more optimized IO path and performs flushing on write. For 128k reads and writes it almost matches the performance of the physical disk, but for 4k operations the higher transaction rate highlights its overheads.

Figure 9 shows 99th percentile latency for block IO with a queue depth of 1. Here Firecracker fares fairly well for small blocks: 4kB reads are only 49µs slower than native reads. Large blocks have significantly more overhead, with Firecracker more than doubling the IO latency. Writes show divergence in the implementation between Firecracker and Cloud Hypervisor with the latter having significantly lower write latency. The write latency of Firecracker and Cloud Hypervisor have also be considered with care since neither supports flush on write.

Tests with multiple MicroVMs showed no significant contention: running multiple tests each with a dedicated disk, both Firecracker and QEMU saw no degradation relative to a single MicroVM.

Network performance tests were run with *iperf3*, measuring bandwidth to and from the local host over a *tap* interface with 1500 byte MTU. We measured both the performance of a single stream as well as 10 concurrent streams. The results are summarised in Table 1. The host (via the tap interface) can achieve around 44Gb/s for a single stream and 46Gb/s for 10 concurrent streams. Firecracker only achives around 15Gb/s for all scenarios while Cloud Hypervisor achieves slight higher performance, likely due to a slight more optimised virtio implementation. The QEMU throughput is roughly the same as for Could Hypervisor. While Firecracker has a little lower throughput than the other VMMs we have not seen this to be a limitation in our production environment.

As with block IO, performance scaled well, with up to 16 concurrent MicroVMs able to achieve the same bandwidth.

VMM	1 RX	1 TX	10 RX	10 TX
loopback	44.14	44.14	46.92	46.92
FC	15.61	14.15	15.13	14.87
Cloud HV	23.12	20.96	22.53	N/A
Qemu	23.76	20.43	19.30	30.43

Table 1: `iperf3` throughput in Gb/s for receiving (RX) in the VM and transmitting (TX) from the VM for a single and ten concurrent TCP streams.

We expect that there are significant improvements still possible to increase latency and throughput for both disk IO and networking. Some improvements, such as exposing parallel disk IOs to the underlying storage devices, are likely to offer significantly improved performance. Nevertheless, the virtio-based approach we have taken with Firecracker will not yield the near-bare-metal performance offered by PCI pass-through (used in our standard EC2 instances); hardware is not yet up to the task of supporting thousands of ephemeral VMs.

5.4 Does Firecracker Achieve Its Goals?

Using the six criteria from Section 2, we found that while there is scope for improving Firecracker (as there is scope for improving all software), it does meet our goals. We have been running Lambda workloads in production on Firecracker since 2018.

Isolation: The use of virtualization, along with side-channel hardening, implementation in Rust, extensive testing and validation makes us confident to run multiple workloads from multiple tenants on the same hardware with Firecracker.

Overhead and Density: Firecracker is able to run thousands of MicroVMs on the same hardware, with overhead as low a 3% on memory and negligible overhead on CPU.

Performance Block IO and network performance have some scope for improvement, but are adequate for the needs of Lambda and Fargate.

Compatibility: Firecracker MicroVMs run an unmodified (although minimal) Linux kernel and userland. We have not found software that does not run in a MicroVM, other than software with specific hardware requirements.

Fast Switching: Firecracker MicroVMs boot with a production kernel configuration and userland in as little as 150ms, and multiple MicroVMs can be started at the same time without contention.

Soft Allocation: We have tested memory and CPU oversubscription ratios of over 20x, and run in production with ratios as high as 10x, with no issues.

In response to this success, we have deployed Firecracker in production in AWS Lambda, where it is being used successfully to process trillions of events per month for millions of different workloads.

6 Conclusion

In building Firecracker, we set out to create a VMM optimized for serverless and container workloads. We have successfully deployed Firecracker to production in Lambda and Fargate, where it has met our goals on performance, density and security. In addition to the short-term success, Firecracker will be the basis for future investments and improvements in the virtualization space, including exploring new areas for virtualization technology. We are excited to see Firecracker being picked up by the container community, and believe that there is a great opportunity to move from Linux containers to virtualization as the standard for container isolation across the industry.

The future is bright for MicroVMs, both in and out of the cloud. Challenges remain in further optimizing performance and density, building schedulers than can take advantage of the unique capabilities of MicroVM-based isolation, and in exploring alternative operating systems and programming models for serverless computing. We expect that there is much fruitful research to do at the VMM and hypervisor levels. Directions we are interested in include: increasing density (especially memory deduplication) without sacrificing isolation against architectural and microarchitectural side-channel attacks; compute hardware optimized for high-density multi-tenancy; high-density host bypass for networking, storage and accelerator hardware; reducing the size of the virtualization trusted compute base; and dramatically reducing startup and workload switching costs. The hardware, user expectations, and threat landscape around running multitenant container and function workloads are changing fast. Perhaps faster than at any other point in the last decade. We are excited to continue to work with the research and open source communities to meet these challenges.

7 Acknowledgements

The Firecracker team at AWS, and the open source community, made Firecracker and this paper possible. Production experience and feedback from the AWS Lambda and Fargate teams was invaluable. Thanks to Tim Harris, Andy Warfield, Radu Weiss and David R. Richardson for their valuable feedback on the paper.

References

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat

Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. *IACR Cryptology ePrint Archive*, 2018:1060, 2018.

[3] Jens Axboe. Fio: Flexible i/o tester, 2019. URL: https://github.com/axboe/fio/.

[4] Microsoft Azure. Azure functions, 2019. URL: https://azure.microsoft.com/en-us/services/functions/.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[6] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 239–252, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2465351.2465375, doi:10.1145/2465351.2465375.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[8] Paolo Bonzini. Minimal x86 firmware for booting linux kernels, 2019. URL: https://github.com/bonzini/qboot.

[9] Reto Buerki and Adrian-Ken Rueegsegger. Muen-an x86/64 separation kernel for high assurance. *University of Applied Sciences Rapperswil (HSR), Tech. Rep*, 2013.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. *arXiv preprint arXiv:1811.05441*, 2018.

[11] Google Cloud. KNative, 2018. URL: https://cloud.google.com/knative/.

[12] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association. URL: https://www.usenix.org/conference/atc19/presentation/fouladi.

[13] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, pages 363–376, 2017.

[14] The Operstack Foundation. Kata Containers - The speed of containers, the security of VMs, 2017. URL: https://katacontainers.io/.

[15] Google. gvisor: Container runtime sandbox, November 2018. URL: https://github.com/google/gvisor.

[16] Google. Google cloud functions, 2019. URL: https://cloud.google.com/functions/.

[17] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM. URL: http://doi.acm.org/10.1145/74850.74870, doi:10.1145/74850.74870.

[18] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[19] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page cache attacks. *arXiv preprint arXiv:1901.01161*, 2019. URL: https://arxiv.org/abs/1901.01161.

[20] Vipul Gupta, Swanand Kadhe, Thomas Courtade, Michael W. Mahoney, and Kannan Ramchandran. Oversketched newton: Fast convex optimization for serverless systems, 2019. arXiv:1903.08857.

[21] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[22] Apache HttpClient. Thread interrupt flag leaking when aborting httprequest during connection leasing stage, 2019. URL: https://issues.apache.org/jira/browse/HTTPCLIENT-1958.

[23] Apache HttpCore. Connection leak when aborting httprequest after connection leasing, 2019. URL: https://issues.apache.org/jira/browse/HTTPCORE-567.

[24] Intel. Nemu: Modern hypervisor for the cloud, 2018. URL: https://github.com/intel/nemu.

[25] Intel. Cloud hypervisor, 2019. URL: https://github.com/intel/cloud-hypervisor.

[26] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

[28] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, June 2014. URL: https://doi.org/10.1145/2678373.2665726, doi:10.1145/2678373.2665726.

[29] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: The linux virtual machine monitor. In *In Proc. 2007 Ottawa Linux Symposium (OLS '07)*, 2007.

[30] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[31] James Larisch, James Mickens, and Eddie Kohler. Alto: lightweight vms using virtualization-aware managed runtimes. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, page 8. ACM, 2018.

[32] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 1–13, Santa Clara, CA, 2017. USENIX Association. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/li-yiwen.

[33] Arm Limited. Cache speculation side-channels, version 2.4. Technical report, October 2018. URL: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper.

[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[35] John DC Little. A proof for the queuing formula: L = λ w. *Operations research*, 9(3):383–387, 1961.

[36] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 1. ACM, 2016.

[37] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122. ACM, 2018.

[38] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3132747.3132763, doi:10.1145/3132747.3132763.

[39] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019. URL: http://arxiv.org/abs/1902.05178, arXiv:1902.05178.

[40] OASIS. Virtual i/o device (virtio) version 1.0, March 2016.

[41] OpenFaaS. OpenFaaS: Serverless Functions Made Simple , 2019. URL: https://www.openfaas.com/.

[42] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag. URL: http://dx.doi.org/10.1007/11605805_1, doi:10.1007/11605805_1.

[43] K. Park, S. Baeg, S. Wen, and R. Wong. Active-precharge hammering on a row induced failure in ddr3 sdrams under 3nm technology. In *2014 IEEE International Integrated Reliability Workshop Final Report (IIRW)*, pages 82–85, Oct 2014. doi:10.1109/IIRW.2014.7049516.

[44] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud function services, 2019. arXiv:1911.11727.

[45] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. *SIGARCH Comput. Archit. News*, 39(1):291–304, March 2011. URL: http://doi.acm.org/10.1145/1961295.1950399, doi:10.1145/1961295.1950399.

[46] The Chromium Projects. Site isolation design, 2018. URL: https://www.chromium.org/developers/design-documents/site-isolation.

[47] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, 2019. USENIX Association. URL: https://www.usenix.org/conference/nsdi19/presentation/pu.

[48] Rusty Russell. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. URL: http://doi.acm.org/10.1145/1400097.1400108, doi:10.1145/1400097.1400108.

[49] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi.org/10.1145/3319535.3354252, doi:10.1145/3319535.3354252.

[50] Amazon Web Services. AWS Fargate - Run containers without managing servers or clusters, 2018. URL: https://aws.amazon.com/fargate/.

[51] Amazon Web Services. Aws lambda - serverless compute, 2018. URL: https://aws.amazon.com/lambda/.

[52] Amazon Web Services. Firecracker production host setup recommendations, 2018. URL: https://github.com/firecracker-microvm/firecracker/blob/master/docs/prod-host-setup.md.

[53] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra, 2018. arXiv:1810.09679.

[54] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels, 2018. arXiv:1806.07480.

[55] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1755913.1755935, doi:10.1145/1755913.1755935.

[56] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2592798.2592812, doi:10.1145/2592798.2592812.

[57] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern linux api usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 16:1–16:16, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2901318.2901341, doi:10.1145/2901318.2901341.

[58] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, Technical report, 2018.

[59] Dan Williams and Ricardo Koller. Unikernel monitors: Extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver,

CO, 2016. USENIX Association. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams.

[60] Dan Williams, Ricardo Koller, and Brandon Lum. Say goodbye to virtualization for a safer cloud. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.

[61] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

Rex: Preventing Bugs and Misconfiguration in Large Services Using Correlated Change Analysis

Sonu Mehta[1], Ranjita Bhagwan[1], Rahul Kumar[1], Chetan Bansal[1], Chandra Maddila[1], B. Ashok[1], Sumit Asthana[1], Christian Bird[2], and Aditya Kumar[1]

[1]Microsoft Research India
[2]Microsoft Research Redmond

Abstract

Large services experience extremely frequent changes to code and configuration. In many cases, these changes are correlated across files. For example, an engineer introduces a new feature following which they also change a configuration file to enable the feature only on a small number of experimental machines. This example captures only one of numerous types of correlations that emerge organically in large services. Unfortunately, in almost all such cases, no documentation or specification guides engineers on how to make correlated changes and they often miss making them. Such misses can be vastly disruptive to the service.

We have designed and deployed Rex, a tool that, using a combination of machine learning and program analysis, learns *change-rules* that capture such correlations. When an engineer changes only a subset of files in a change-rule, Rex suggests additional changes to the engineer based on the change-rule. Rex has been deployed for 14 months on 360 repositories within Microsoft that hold code and configuration for services such as Office 365 and Azure. Rex has so far positively affected 4926 changes without which, at the very least, code-quality would have degraded and, in some cases, the service would have been severely disrupted.

1 Introduction

Large-scale services run on a foundation of very large codebases and configuration repositories. To run uninterrupted, a service not only depends on correct code, but also on correct network and security configuration, and suitable deployment specification. This causes various dependencies both within and across components/sources of the service which emerge organically. When an engineer changes a certain region of code or configuration, these dependencies require them to make changes to other code or configuration regions. For instance, when an engineer adds a new feature to a service, they may need to add a function to test the feature. Also, they may need to configure the service to deploy the new feature only to a small set of machines to test it further. Similarly, when an engineer renames a service API, they must also change firewall rule specifications so that the rules apply to the now renamed API rather than to the old one.

Such correlations can occur between code files across components, between code and configuration files, or between configuration files. Unfortunately, unlike pure code, which goes through compilation, reviewing and systematic testing to weed out bugs, these correlations are often not specified, checked for, and are left undocumented. Consequently engineers, with no documentation or specification to go by, often miss making necessary changes to code or configuration files. This can delay deployment, increase security risks and, in some cases, even disrupt the service completely. Disruptions due to such correlations are surprisingly frequent [12]. For instance, an engineer recently caused a disruption at Salesforce because they did not perform all necessary dependent configuration changes related to a change they initiated [22].

To address this problem, we present Rex, a tool that learns these correlations using a combination of machine learning and program analysis. Using *association rule mining* on many months of file changes, Rex determines sets of files that often change together. Rex also uses *differential syntax analysis* to learn *change-rules*: each change-rule captures a set of correlated changes across files. When an engineer makes a file change, Rex analyzes the change and uses the change-rules to suggest additional changes if required.

While the idea of using association rule mining to determine correlations in code and configuration has been proposed before [6, 35, 37], previous work has not concentrated on generalizing the algorithm. To the best of our knowledge, Rex is the first tool that combines association rule mining with syntactic analysis to determine change-rules. Moreover, Rex takes the crucial step of making correlated change analysis generalize well to multiple file-types and services, and deploying it at a large-scale. We do this through three key observations made by studying the characteristics of services:

1. Correlations occur in a multitude of unpredictable ways. Consequently, Rex's algorithm should not rely on any hard-coded domain knowledge, neither should it depend on any manual configuration or tuning.

2. Configuration management practice varies widely across services and projects. Every service has distinct configuration management and maintenance strategies as a result of which machine learning models have to be service or project-specific, with no extrapolation from one to the other. To make matters even more challenging, even a single service or project can change characteristics significantly over time. Hence, Rex's models have to be periodically retrained so that its suggestions can be accurate.

3. Care has to be taken while applying association rule mining on large code and configuration files. Services depend upon a large amount of code and configuration. Applying rule mining which is exponential in the size of the input at the level of individual code and configuration constructs is simply not feasible. We realized this early in the design process and therefore apply rule-mining at the *file-level*.

Rex is deployed on 360 Microsoft repositories which hold code and configuration for services such as Exchange Online, OneDrive, Azure, Dynamics CRM and Skype. We are currently scaling out Rex at a fast pace, on-boarding almost one repository per day. Till date, Rex has suggested 4926 changes to engineers that, if not made, may have adversely affected our services in many ways.

In this paper, we make the following contributions:

- We demonstrate different types of correlations that exist across code and configuration of large services.
- We describe a novel two-step algorithm to perform correlated change analysis involving file-level association rule mining followed by differential syntactic analysis of the changes made to the files.
- We have implemented and deployed Rex and provide an evaluation based on our deployments.
- We have performed an extensive user study to understand how useful Rex is in practice.

Section 2 describes different types of correlations Rex has found across many services. Section 3 provides an overview of Rex's approach, limitations, and challenges. Section 4 explains the algorithms Rex uses to suggest changes. Section 5 and Section 6 provide specifics on its implementation and deployment. Finally, Sections 7 and 8 describe a thorough evaluation and user study respectively.

2 Reasons for Correlated Change

Correlations occur due to various reasons. In this section, we describe several categories of correlations we found through our deployments. Table 1 shows a sample of correlated changes that engineers missed making and Rex flagged at commit-time. We note that though these examples are specific to our deployments, the problem of correlated configuration is generic and extends to other organizations as well [14, 22, 27]. We now describe these categories of correlations with the help of the examples in Table 1.

2.1 Flighting

When an engineer adds a new feature, they use canary-testing or "flighting" to deploy it in stages. They first deploy it to a small subset of machines to ensure that the feature works as planned and does not cause disruptions. Once they ensure this, they deploy the feature more widely. Hence, when the engineer adds code to implement a new feature, they also need to add configuration to files that define the set of machines that will test this feature. Services implement flighting in many different ways. Example 3 shows an instance where the engineer who develops the new feature decides which set of machines to run the feature on. Example 7, for a different service, shows an instance of a change where the engineer who develops the new feature does not directly turn on the feature: they provide a "code switch" which other engineers can use to turn on flighting. These two examples again illustrate why Rex needs to learn such varied change-rules from data and why rule-based engines would not work across services.

2.2 Replicating Code and Configuration

While clearly not recommended, we find that engineers sometimes replicate files and file contents across different logical boundaries of the service. They do this since, without replication, there will be a larger number of dependencies across files and components. This in turn will lead to less modular code-bases which may take longer to test, debug, and deploy. Example 2 shows an instance where a configuration file is replicated across different alerting frameworks. An engineer changed one, without knowing that a replica existed within the other alerting framework. Rex flagged this file and the engineer immediately changed the other file as well.

2.3 Complex Configuration

Configuring services is a complex task and, as a result, several correlations show up between configuration files. Example 4 shows an instance where an engineer renamed a microservice, but forgot to change the name of the service in the file that contained its firewall rules. This could have caused a security issue. Example 8 shows another instance where hardware configuration files are correlated, and missing this change could have caused a service disruption.

No.	File 1	File 2	Reason	If File 2 unchanged
1	Source code (JS)	Test file (JS)	An engineer added new functionality to source-code and needed to add a unit-test to test this functionality to another file.	Without the test, a **bug in File 1 may go unnoticed**.
2	Component definitions (C#)	Component definitions (C#)	An engineer changed a parameter in the definition of a set of components in File 1. An alerting framework uses File 1 to determine which components to probe and alert on. File 2, very similar to File 1, is used by a different alerting framework for the same reason. Hence, the engineer had to make the same changes to File 2 as well.	The second alerting framework would malfunction, leading to **incorrectly suppressed alerts**.
3	Feature set definitions (INI)	Flight definition (INI)	An engineer added a new feature to a service. She enabled the feature in deployment by updating a settings file. Additionally, she had to specify how to "flight" the change, i.e., which subset of machines to run the new feature on. File 2 contains these flight configurations for every feature.	The **feature will not deploy** until the engineer changes File 2.
4	Microservice Registry (XML)	Firewall rule definitions (XML)	An engineer changed a microservice's name in the microservice registry. File 2 holds a mapping from a set of microservices to the firewall rules that apply to them, so the engineer needed to change the microservice's name in File 2 as well.	The required firewall rules would not have applied to the renamed microservice, causing a **security threat**.
5	Shell script (PS1)	File storing security vulnerabilities (XML)	File 2 keeps a record of potential security vulnerabilities such as cleartext passwords and code susceptible to injection attacks. It stores a record with both the offending file name, and the line number where the vulnerability exists. A security scanner uses File 2 to *ignore* the vulnerabilities it specifies, to avoid flagging the same vulnerabilities repeatedly. Hence, when an engineer adds or removes lines from File 1, they need to appropriately change the line number of the vulnerable code in File 2 as well.	The security scanner would ignore a completely different, potentially vulnerable, line of code. This could cause a **security threat**.
6	Shell script (PS1)	Shell script (PS1)	File 1 defines a function, File 2 calls it. The engineer made a change to the function name and parameters in File 1, so they had to change how File 2 calls the function. If this code were compiled rather than interpreted, the compiler would have caught the error.	The scripts determine how the service is deployed, and hence this error would have caused **deployment to fail**.
7	Style sheet file (SCSS)	Flight definitions file (C#)	An engineer made a web-design change in File 1, and wanted to flight it on a small set of machines. File 2 contains definitions of "code switches" that engineers can use to turn on the new design change.	Without an appropriate code switch, engineers cannot turn on the design change, and hence, this would have caused unnecessary **deployment delays** of the new look.
8	Config file maintaining SKU information of machines in a Data center (XML)	File maintaining rack definitions for the data center (XML)	Operators update File 1 when a new set of machines with a new SKU is introduced to the data center. If the new machines are deployed, also need to update File 2 which specifies which machines sit in each rack.	Several other functions use these configuration files. Incorrect data center configuration can cause faulty functioning of the service and hence **disruption**.

Table 1: This table describes some real examples of correlated changes that engineers missed and Rex flagged in our deployments. The *Reason* column captures why the two files are correlated. The last column describes what may have happened, had Rex not flagged the issue and notified the engineer.

2.4 Testing

Example 1 shows that, when an engineer adds a new feature to code, they should consider adding a new test for that feature in a separate file that contains only tests. While this is fairly common across multiple code-bases and services, each code-base has its own organization structure for separating test code from the main production code. Rex automatically detects such structures without any manual input.

2.5 Scripting

Often, administrators use scripts to test and deploy services. These scripts can have complex inter-dependencies which, unlike compiled code, can go unchecked at commit-time. For instance, in Example 6, an engineer changed function definition in one script and hence they had to change the way the function was called in another script. Rex caught this issue, while existing IDEs and compilers could not.

2.6 Miscellaneous

Apart from the categories we have mentioned so far, Rex also flags somewhat rare cases of correlation which can have high impact. In Example 5, File 2 maintains a list of line-numbers of vulnerable code across different files in the code-base. The idea is to maintain a record of all vulnerabilities that have already been found and vetted by engineers. Thus, when an engineer adds *n* lines of code to File 1, they also changed the line number of the vulnerable code in File 1. Hence they need to increment the line number in File 2 by *n*. While such categories of correlations are rare, we notice multiple such rare cases. This further confirms the value of using a learning-based approach.

Note that, for simplicity, the table shows examples that involved only two files. In reality, change-rules can contain more than two files. Moreover, the correlations for similar tasks are very different for different services. Example 3 and Example 7 in Section 2.1 talks about two different ways of flighting a feature. Even within a service, the correlations are dynamic and keep changing with time. We believe no existing syntactic or semantic analysis techniques or heuristic based model could have effectively and efficiently captured such diverse and complex correlations.

3 Problem Overview

In this section, we define the problem that Rex solves, the approach to it, and describe some limitations of the approach. Finally, we lay out the challenges we faced as we designed and deployed Rex.

3.1 Approach

Rex applies association rule mining on months of commit logs to find correlated changes. Association rule mining is fundamentally an exponential algorithm. Finding correlations between individual configuration parameters and code constructs such as variables and functions will be prohibitively expensive simply because of the sheer large numbers of such constructs [28, 33, 35]. Hence to scale well, we decided to mine change-rules *at the file-level*. While the approach is coarse-grained and does not capture correlations perfectly, it makes the solution practical to deploy at a large scale.

Rex learns change-rules in two steps: *change-rule discovery* and *change-rule refinement*. In the discovery step, it uses association rule mining to find sets of files that change together "frequently". A set of parameters determine how frequently the files need to change for Rex to learn the change-rule. Section 4.2 provides more detail on this algorithm and Section 4.5 shows how we tune its parameters to maintain effectiveness through changing characteristics of the service.

After change-rule discovery, Rex runs the second step, namely, change-rule refinement. The idea is to make each change-rule, which is coarse-grained and at the file-level, more precise. Rex analyzes the change in every file of the change-rule to determine what *types* of changes are correlated. Section 4.3 describes this procedure further. Finally, Rex makes suggestions to engineers based on the learnt rules.

3.2 Design Goals

Rex's design is driven by two factors. First, it needs to be *generic*: its techniques need to work well across file-types, service-types, and programming languages. Second, it needs to be *effective*: it should find subtle misconfigurations and bugs which existing tools cannot catch. To achieve these goals, our solution has the following characteristics:

No Manual Inputs: The main goal of Rex is to help engineers find misconfigurations and bugs early, while minimally intruding upon on their already busy schedule. We therefore design it to work with existing systems and logs, and do not require any additional logging or inputs from the engineers. We believe this is one of the main reasons that Rex is being adopted widely across our organization over multiple services.

Correlation, not causation: Rex flags correlations, and does not detect causality because the cause of a specific set of correlated changes may not be captured by any logs. For instance, consider Example 2 in Table 1: changing one file of component definitions does not *cause* the change in the other. An engineer was extending the alerting infrastructure to a larger number of components, and this caused the need to change both files.

3.3 Scope

As with any machine learning-based approach, Rex is a best-effort service. Sometimes it may miss suggesting required changes (false-negatives) and conversely, it also suggests changes when none are needed (false-positives). As we describe in Section 4, we tune Rex so that it catches as many misconfigurations as possible even though this may come at the cost of a higher number of false-positives. Take for instance Example 5. We need to change File 2 only if the line-number of a vulnerable code-snippet in File 1 changes. It is fundamentally difficult for a generic technique to learn the specific semantics of this particular correlation. Rather, Rex suggests that the engineer change File 2 whenever they change File 1, even if the line-numbers in File 1 do not change. Such a suggestion will be a false-positive.

3.4 Challenges

Determining the right set of correlations has several challenges associated with it.

Imperfect Ground-truth: The largest challenge we faced as we designed Rex was imperfect ground-truth. The reasons for this are many. First, correlations are often subtle and do not necessarily cause compile errors, deployment failures, or immediate service downtime. Consider the issue in Table 1, Example 1, where the engineer needs to add a test for a newly added feature. This is not strictly necessary but definitely recommended. However engineers are often hard-pressed to commit and deploy fast and therefore may not add the test. Hence the commit logs that Rex uses may not always see the two files with the added feature and test changed together. As a consequence, Rex may not learn the change-rule that includes these two files.

Performance: Rex currently runs on 360 repositories, and its adoption is increasing rapidly. Hence we need to ensure there

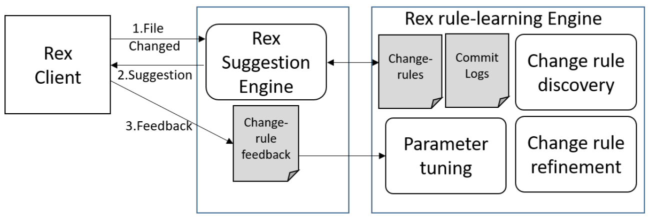


Figure 1: Rex system design.

are no manual steps involved. Additionally, we need to ensure that the rule mining algorithm does not become prohibitively expensive.

4 System Design

In this section, we provide an overview of the different components of Rex. We then describe each component in detail.

4.1 Design Overview

Figure 1 shows an overview of the Rex design. The Rex rule-learning engine periodically learns change-rules that capture which files change together and how. It uses several months of commit logs to do this. For each commit, the commit log contains information about which files changed, and how they changed. Rex's rule-learning engine runs two processes to learn rules: change-rule discovery (Section 4.2) and change-rule refinement (Section 4.3).

The Rex suggestion engine interfaces between the client that uses Rex and the rule-learning engine. When an engineer changes a file, the Rex client notifies the suggestion engine of the change. The suggestion engine looks up applicable change-rules to determine if the engineer may have missed changing a correlated file. If so, the suggestion engine suggests the additional file change back to the client. Our current implementation of the Rex client is built for various source control systems such as Git [29]. It adds suggestions as pull-request comments, whenever required, after every commit in a pull-request. More details on this are in Section 5.

When the Rex client provides the suggestion back to the engineer, they either accept the suggestion by editing the suggested file or not. Rex uses this behavior as feedback to the rule-learning engine. Using this implicit feedback, Rex automatically tunes parameters used to learn the change-rules. Section 4.5 provides more details on the tuning module and why this is essential to scale Rex across hundreds of repositories. Very few engineers provide explicit feedback by replying or resolving the comment and we do not use this because such feedback is very limited and is inherently biased towards negative examples.

```
/…/ServiceLibrary/package.xml =>
/…/ServiceLibrary/src/NuGetPackage/NuGetPackage.nuproj
```

a) Example rule that requires a version number increment in the suggested file.

```
/…/configuration/dailyRun.xml =>
/…/configuration/dogfoodRun.xml
```

b) Example rule that learns nearly identical configuration files.

```
/…/Framework/src/Actions/RunDailyWorkflows.cs, /…/Config/Engine.Settings.ini
=> /…/Config/DWEngine.flight.ini
```

c) Example rule that learns that engineer added code AND added configuration used by the code, so they need to specify how to flight this code.

Figure 2: Some example rules from the change-rule discovery step. Note that rules are not limited to only file pairs. Example c) shows an example where two files are learned on the LHS.

4.2 Change-rule Discovery

In this section, we describe the first-step towards learning rules, which is change-rule discovery. We use six months of commit data for rule-mining. First, Rex prunes the commit logs to exclude commits that are aggregates of smaller commits caused by merging branches (squashed changes), or porting a set of commits across branches. Since these commits put together a set of smaller commits that may not have any relation with each other, they do not capture true correlations between files. Moreover, such large commits make mining rules prohibitively expensive. Figure 2 shows some examples of rules that Rex has learned.

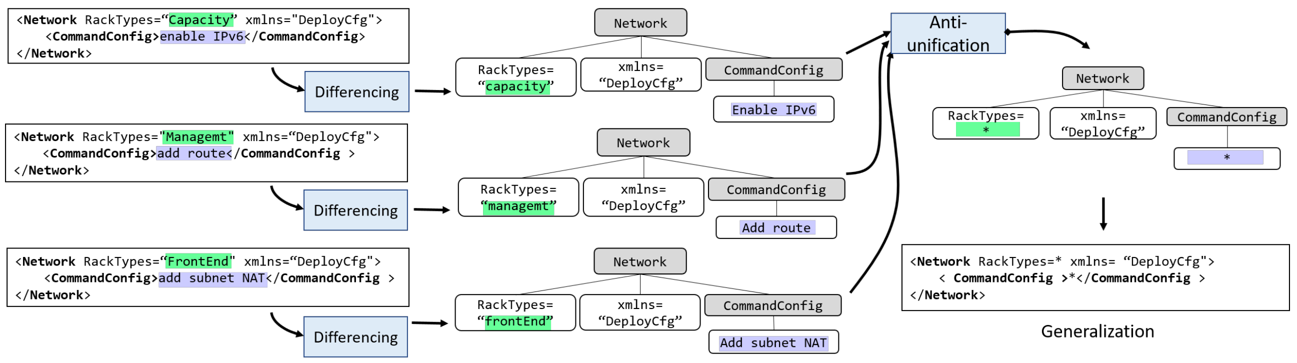
Rex runs the rule mining algorithm considering each commit as a transaction. First, it discovers *frequent item-sets* using the FP-Growth algorithm [13]. A frequent item-set is a set of files that change together very often. Mathematically, we define a frequent item-set as $\mathbf{F} = \{f_1, \ldots, f_n\}$ where files f_1 through f_n have changed together at least s_{min} times. s_{min} is the *minimum support* defined for the model. The *support* of the frequent item-set, $s_\mathbf{F}$, is defined as the number of times files f_1 through f_n change together. Hence, $s_\mathbf{F} \geq s_{min}$.

Next, the algorithm generates change-rules from frequent item-sets. From the frequent item-set $\mathbf{F}$, Rex learns the rule $\mathbf{X} \implies \mathbf{Y}$ such that $\mathbf{X} \subset \mathbf{F}, \mathbf{Y} \subset \mathbf{F}, \mathbf{X} \cap \mathbf{Y} = \phi, \mathbf{X} \cup \mathbf{Y} = \mathbf{F}$.

The *confidence* of the rule is the number of times all the files in $\mathbf{F}$ change together (support of file-set $\mathbf{F}$) divided by the number of times all the files in $\mathbf{X}$ change together (support of file-set $\mathbf{X}$). The rule's confidence is therefore $s_{\mathbf{X}\cup\mathbf{Y}}/s_\mathbf{X}$. Hence, the more often files in sets $\mathbf{X}$ and $\mathbf{Y}$ change together, the higher the confidence of the rule. Rex learns a rule only if it has confidence above a minimum confidence c_{min}.

4.3 Change-rule Refinement

In this Section, we describe the change-rule refinement process. Currently our implementation supports configuration files, but it can be extended to support code files as well. In our description, we concentrate on `xml` files though the same



```
<Network RackTypes="Capacity" xmlns="DeployCfg">
    <CommandConfig>enable IPv6</CommandConfig>
</Network>
```

```
<Network RackTypes="Managemt" xmlns="DeployCfg">
    <CommandConfig>add route</CommandConfig >
</Network>
```

```
<Network RackTypes="FrontEnd" xmlns="DeployCfg">
    <CommandConfig>add subnet NAT</CommandConfig >
</Network>
```

```
<Network RackTypes=* xmlns= "DeployCfg">
    < CommandConfig >*</CommandConfig >
</Network>
```

Figure 3: Steps of change-rule refinement for a rule `network_dc1.xml` $\implies$ `network_dc2.xml`. Three separate commits are made to a single configuration file. Each one adds an XML attribute `Network`, but with different values. From each of these three, Rex learns difference trees that codify the additions. All these difference trees are input to the anti-unification algorithm which outputs a generalization for this type of addition.

techniques apply to other file-types such as `json`.

Consider the following examples which have arisen in our deployments:

Ex 1. Network configuration: An engineer adds new commands to a file `NetConfig_dc1.xml` that configure racks in data center 1. These changes need to be applied to all data centers, and hence, the engineer has to change similar configuration files for other data centers as well, say `NetConfig_dc2.xml`. Rex should therefore suggest these changes if the engineer does not make them. However, in many cases, the engineer makes changes to `NetConfig_dc1.xml` that apply only to data center 1 and not to data center 2. For instance, they may add a new subnet only to data center 1. In this case, Rex should not suggest changing `NetConfig_dc2.xml`. Change-rule discovery alone cannot differentiate these two scenarios.

Ex 2. Role-based access control: Several of our services implement role-based access control. Engineers often define a new role in a file `RoleDefn.xml`. When they do so, they should also change `RoleMembership.xml`, which specifies the users or groups that are associated with the new role. However, if the engineer is only modifying an already existing role definition in `RoleDefn.xml`, they need not change `RoleMembership.xml`.

These examples show that, in some cases, for a rule $\mathbf{X} \implies \mathbf{Y}$, $\mathbf{Y}$ changes only if $\mathbf{X}$ changes in a specific way. While for code, compilers often catch such correlations and dependencies, configuration files lack an equivalent safety net.

Change-rule refinement has two parts. First, given a configuration file x_C, it learns all *generalizations* of additions, deletions and modifications made to x_C, where a generalization is in the form of a regular expression. Next, for any change-rule $\mathbf{X} \implies \mathbf{Y}$ already learned by change-rule discovery where $x_C \in \mathbf{X}$, it refines the rule further. We now describe these two steps in detail.

4.3.1 Learning generalizations

Figure 3 shows an example of this. Rex creates a set of all commits $\mathbf{C}$ that modify x_C. For every commit in $\mathbf{C}$, Rex computes a syntactic difference between the old and new version of x_C. To do this, Rex constructs *parse trees* for both versions, and then uses a novel differencing algorithm to compute the difference between the two parse trees, which we call a *difference tree*. For example, in Figure 3, three changes were added to x_C in three different commits. Each change added an XML node named `network`, but with varying attribute values. In each case, Rex's differencing algorithm outputs a difference tree capturing the difference. The shaded vertices are XML nodes, while the unshaded vertices are XML attributes.

Next, from the difference trees, Rex learns generalizations of the changes that happen to the configuration file. To extract these generalizations, Rex uses the process of *anti-unification* [15, 23]. The anti-unification algorithm learns regular expressions that are the *most specific generalizations* of the difference trees. In each of the three changes shown in Figure 3, the `xml` attribute `RackTypes` has different values. The `xml` attribute `CommandConfig` too has different text values. Taking the three difference trees as input, the anti-unification algorithm outputs the generalized difference tree, and thereby the most specific generalization of the three changes.

While Figure 3 describes one example generalization, a file x_C may have many more such generalizations. Rex learns all such generalizations of additions, deletions and modifications to the configuration file x_C. Say this set is $G(x_C) = \{g_1, g_2, \ldots, g_n\}$.

4.3.2 Refining Rules

Next, given a rule learned during change-rule discovery $\mathbf{X} \implies \mathbf{Y}$, where $x_C \in \mathbf{X}$, Rex learns more fine-grained rules of the form $\{\mathbf{X}, (x_C, g_i)\} \implies \mathbf{Y}, g_i \in G(x_C)$. This rule says that when all files in $\mathbf{X}$ change, Rex will suggest changing $\mathbf{Y}$

only if the change to file x_C matches generalization g_i.

This is done in the following way. Say the number of times a change in file x_C matches g_i *and* all files in $\mathbf{Y}$ change is n. Conversely, say the number of times a change in file x_C matches g_i and files in $\mathbf{Y}$ *do not* change is $\tilde{n}$. If $n/(n+\tilde{n}) > t$, where t is a threshold we call the refinement threshold, Rex refines the rule by adding the tuple (x_C, g_i) to the left-hand side of the rule. This means that Rex now makes the suggestion only if the change to x_C matches the regular expression g_i. Thus, change-rule refinement cuts down on false-positive suggestions. In all our deployments, we set t to 0.75.

Though our implementation of the differencing algorithm is specific to configuration files, we can also extend this to code files. The code differencing algorithm could learn syntactic features such as “function added", “if-condition changed", etc. We could refine rules for code using these features. Based on a careful empirical study we conducted while going through true-positives and false-positives that change-rule discovery generated, we observed that a lot of issues with code files are already addressed by compilers. So, we do not see many false-positives for code files when the engineer has commited changes, because in most cases, engineers commit changes after compiling the code. This will become more clear in the next section 4.4 where we describe how these rules are used. Rex uses these rules to make recommendations for missing files after an engineer has committed a change. Such a tool for code files would be helpful for developers if the suggestions are made at IDE (Integrated Development Environment) level. We leave this for future-work.

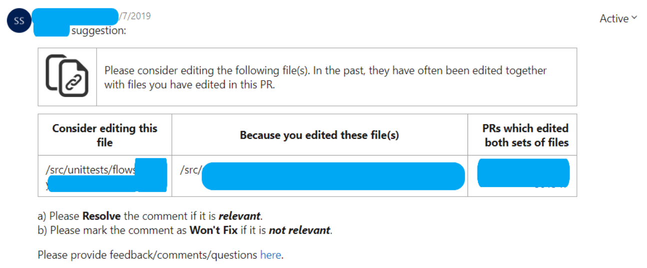
4.4 Suggestion Engine

In this Section, we describe how the Rex suggestion engine uses the rules learned by change-rule discovery and change-rule refinement.

When an engineer commits a code or configuration change, the Rex client calls the suggestion engine which determines the set of rules that match the commit. If there is a match, the suggestion engine checks if any of the files in $\mathbf{Y}$ are unchanged by the commit. If it does find such a file, the suggestion engine recommends that the files be changed. If the engineer does indeed change the suggested files, the suggestion is considered a *true-positive*. Else, the suggestion is a *false-positive*. These numbers are used both for parameter tuning (Section 4.5) and evaluation for Rex (Section 7).

4.5 Parameter Tuning

As we deployed Rex on more projects and services, we noticed that the frequency and the nature of changes varied widely, not just across projects and services, but also within the same project at different times. Hence, once a day, Rex uses the feedback from the suggestion engine to tune models.


SS [redacted]/7/2019 Active
[redacted] suggestion:

Please consider editing the following file(s). In the past, they have often been edited together with files you have edited in this PR.

Consider editing this file	Because you edited these file(s)	PRs which edited both sets of files
/src/unittests/flows[redacted]	/src/[redacted]	[redacted]

a) Please **Resolve** the comment if it is ***relevant***.
b) Please mark the comment as **Won't Fix** if it is ***not relevant***.

Please provide feedback/comments/questions here.

Figure 4: Screenshot of a Rex pull-request comment. Sensitive text has been masked.

Association rule mining has two main tunable parameters, the minimum support s_{min} and the minimum confidence c_{min}. Rex tunes only s_{min} and sets c_{min} to a constant, relatively low value of 0.5. This is because while we want change-rule discovery to learn a relatively large set of rules, perhaps some with low confidence, we use change-rule refinement to make the rules more precise.

We train various models by varying the value of s_{min}. We do not set s_{min} to values less than 4, since that leads to too many rules and slows down rule-mining. We then evaluate each model on one month's data and pick the *best* one using the described approach. We apply the model after every commit[1]. We measure the number of false-positives and true-positives. In addition, we also compute *false-negatives* for a model. This is the number of true-positives that the model with s_{min} set to 4 found, but the current model did not. Hence, we compute every model's false-negatives relative to the model with the lowest value of s_{min}, which learns the largest number of rules.

From these numbers, we can compute precision, recall and F1-score for each model. Finally, we pick the model with the highest F1-score and deploy it.

5 Implementation

Rex is implemented using C# on top of the .NET framework and deployed using a combination of services provided by Microsoft Azure [19]. Rex is currently deployed on 360 repositories across multiple Microsoft services. There are three main components of Rex:

Data Ingestion and Loading: Using Azure DevOps [18] and Github [11] APIs, batch jobs execute at predefined intervals to ingest information about pull requests, commits, files, diffs, etc. for each repository where Rex is enabled. All data is stored using SQL databases. Currently, there is a one-to-one mapping between a repository and a database. The SQL database schema is normalized and allows for efficient querying of commit and file data. Newly onboarded repositories are back filled with 6 months of data.

[1]Our evaluations are GIT-specific, so we apply Rex after every commit to a pull-request. This approach extends to other version control systems as well.

Learning: For each repository, every day a new *model* is learned. Currently, Rex uses the FP-growth algorithm [13] to learn association rules from the six months of commit history in a repository. Rex also analyzes `xml` files more deeply using XmlDiffAndPatch [10] in order to perform change-rule refinement using the anti-unification algorithm [15, 23]. The learned model and metadata about the model is saved in the respective database for the repository; thus, resulting in a *repository-local* model.

Decorator: The *pull-request decorator* performs the functionality of the suggestion engine. It subscribes to events in each repository using APIs. For each pull-request that is created or updated, the decorator mines details on the fly, performs inferencing of the pull-request using the latest model stored in the repository database and creates systematic comments in the pull-request for all valid and new Rex suggestions.

6 Deployment

Deploying Rex is easy: an administrator of a repository only needs to provide a URL of their repositories. Repo admins need not provide any inputs to Rex, they need not configure it, and hence the effort to on-board a repository is minimal. We started deploying Rex with one repository in October 2018. Since then, its adoption has steadily grown. Rex is now deployed on 360 repositories for services such as Exchange Online, OneDrive, Azure, Dynamics CRM, and Skype.

No.	Metrics	Min	Max
1	Total No. of Files	26	99235
2	% of Config Files in Repository	16%	100%
3	Avg.No. of Pull-Requests Per-day	1	279
4	No. of Engineers	3	2885

Table 2: Characteristics of repositories on which Rex runs.

Deployments are on very different types of repositories. Table 2 summarizes the characteristics of the 360 repositories that Rex is deployed on. The characteristics vary widely: we host one of the largest git repositories in the world, while we also host small, relatively inactive repositories. Row 2 captures that our repositories have varying amount of code and configuration. Some repositories hold only configuration information, while others hold mostly code.

6.1 Lessons

In this section, we will outline some lessons and insights we have gathered from these varied deployments. We believe these insights hold in general for tools such as MUVI [16], DeepBugs [20], EnCore [35] and Getafix [24] which use machine learning to flag bugs and misconfigurations.

1. We should distinguish between model precision vs deployment precision. No ML-based tool is perfect, and hence the best way to evaluate it is by observing its precision, which is the ratio of the number of true-positives to the total number of suggestions made. In our implementation, a Rex suggestion is a *true-positive* if, after it is made on a pull-request, the engineer adds the suggested change *within the same pull-request*.

For bug and misconfiguration-detection tools, one needs to compute two types of precision: *model precision* and *deployment precision*. Model precision is the ratio of true-positives to total suggestions that the model makes on test data as opposed to a real deployment. Rex uses the last six months of commit logs as test data. The deployment precision is the ratio of the true positives *actually observed in deployment* to the total number of suggestions shown to engineers.

Invariably, deployment precision is significantly lower than the model precision. This is because Rex provides suggestions only when the engineer makes a Git pull-request. This is after the engineer has had an opportunity to weed out bugs and misconfiguration through subsequent commits made after some unit-testing and reviewing. For instance, say Rex predicts correlations in 100 cases, of which 90 are correct (true-positives) and 10 are incorrect (false-positives). The model precision is thus quite high, i.e. 90%. In actual deployment though, say engineers remember to make the right changes in 88 of the 90 cases Rex discovered. Hence rex shows suggestions only in the 2 remaining cases. On the other hand, Rex does make the same 10 false-positive suggestions. Thus the deployment precision for Rex is $2/(10+2) = 16.7\%$. The deployment precision may seem low, but it is important to note that the suggestions made by Rex are for less obvious correlations which the engineer is unaware of.

2. Flagging high-impact misses offsets the effect of low deployment precision. In the example shown above, the 2 useful suggestions made by Rex in deployment could actually avert severe service disruption. By interviewing several engineers we found that Rex is indeed flagging such issues, and as a result, the engineers consider the low deployment precision acceptable. Therefore when we deploy Rex afresh for a project, we ensure that engineers understand this trade-off and yet appreciate the utility. Also, for this reason, we tune Rex not just for high precision but also for high recall relative to the baseline model, as described in Section 4.5.

Engineers want suggestion interpretability: Engineers like to know why Rex makes a particular suggestion. Therefore, along with every suggestion, we also provide an explanation which shows the past commits from which Rex learned the rule. If an engineer would like to understand the reason for a Rex suggestion, they can view this explanation.

7 Evaluation

Rex has been in deployment for about a year now and has so far found 4926 true-positive suggestions across 360 repositories, many of which have helped avoid severe service outages. In this section, we evaluate Rex. The questions we ask are:

1. *How does change-rule discovery perform?*
2. *What value does change-rule refinement add over change-rule discovery?*
3. *How useful is the parameter-tuning process?*
4. *What is the performance overhead of Rex?*

7.1 Rex Precision

In this section, we evaluate both model precision and deployment precision as we explained in Section 6.1. We first evaluate change-rule discovery.

Table 3 shows the model and deployment precision for change-rule discovery for 7 repositories across 4 services: OneDrive, Azure, Exchange Online and Dynamics CRM. The model was trained on 6 months of commit data and tested on 6 subsequent months of test data. We find that the model precision varies between 66.09% and 82.11%. The deployment precision varies between 6.84% and 16.74%. Notice that a higher model precision does not necessarily imply a higher deployment precision. Consider Azure1 and OneDrive1. Though they both have relatively high model precisions, OneDrive1 has a high deployment precision (16.74%) whereas Azure1's deployment precision is only 6.84%. We believe this variability is due to various reasons specific to the repository, such as the complexity of the repository, the nature of the configuration and the learned rules, etc. Our user study in Section 8 explains this to some extent. We now compare precision for change-rule discovery with and without change-rule refinement (CRR). We use the same train-test split used to evaluate change-rule discovery. We run this experiment only for configuration files written in `xml`, `config`, `csproj`, `proj`, `resx` and `wxs` formats since our differencing algorithm supports them. We do not consider code files. Hence, the deployment precision numbers vary slightly from the results in Table 3.

Table 4 shows the performance of Rex, with and without change-rule refinement, for 6 repositories. In each case, while model precision does improve, the deployment precision improves significantly. For Exchange3, which had a very low deployment precision of 5.03%, the deployment precision with change-rule refinement increased to 18.00%, an increase of about 250%.

7.2 Parameter Tuning

In this section, we justify the importance of tuning the minimum support s_{min} for change-rule discovery both across repositories and also within a single repository.

Tuning across repositories:

The repositories that Rex is deployed on are extremely varied and dynamic, with the number of pull-requests varying

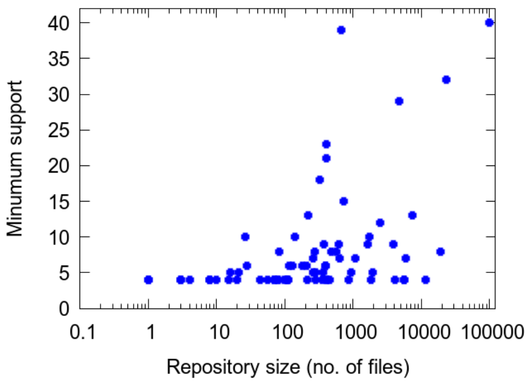


Figure 5: Tuned s_{min} vs repository size

between 1 to 279 per-day. Figure 5 plots, for every repository, the tuned value of minimum support s_{min} against the size of the repository at a given point of time. The size of the repository is the number of files in the repository. While there is a clear correlation of 0.56 between repository size and s_{min}, the repository size by itself does not clearly tell us what the value of s_{min} should be. For instance, for size 700, depending on the repository, s_{min} varies from 4 to 23. This implies that we need to tune s_{min} for each individual repository.

Tuning within a repository: Even within a repository, characteristics change over time. Figures 6a- 6d show the variation of s_{min} with time for four repositories. This variation can be due to multiple reasons. First, decreased commit rates require Rex to lower s_{min} so that a healthy suggestion rate is maintained, even though precision may drop. Second, engineers may add new files to the repository in which case the s_{min} may need to be lowered to learn rules specific to these files.

To understand this fluctuation better, Figures 6e- 6h plot precision, recall and F1-score for one run of the parameter-tuning algorithm for four repositories. Note that the recall here refers to the recall based on the number of true-positive suggestions made by the baseline model with $s_{min} = 4$. The graph shows values of precision, recall and F1-score normalized by the respective values for the baseline model ($s_{min} = 4$). As expected, for all four repositories, as support increases, the model learns fewer rules, and there is a drop in recall. Surprisingly though, as s_{min} increases, precision increases predictably only for Repository 4, and to some extent for Repository 2. Repositories 1 and 3 see some local increases in precision, but overall, it follows a downward-trend.

On analysis of these repositories, we found that majority of true-positives in this case were generated by the rules having low s_{min}. On increasing s_{min}, we do not retain these rules and thus the number of true-positives drop significantly. Even though there is drop in false-positives with the increase in s_{min}, the drop in true-positives is significant to bring the precision value down. This unpredictability in behavior motivates the need to perform regular tuning of Rex models.

Repository	Model Metrics				Deployment Metrics			
	Total	True Positives	False Positives	Precision	Total	True Positives	False Positives	Precision
Exchange1	1869	1342	527	71.80%	519	50	469	9.63%
Exchange2	5216	3659	1557	70.15%	1318	146	1172	11.08%
Exchange3	3634	2635	999	72.51%	932	66	866	7.08%
Azure1	1062	872	190	82.11%	190	13	177	6.84%
OneDrive1	840	672	168	80.00%	221	37	184	16.74%
OneDrive2	367	277	90	75.48%	108	20	88	18.52%
DynamicsCRM1	1666	1101	565	66.09%	490	59	431	12.04%

Table 3: Model and Deployment Precision for Change-Rule Discovery

Repository	Model Metrics			Deployment Metrics		
	Precision (Without CRR)	Precision (With CRR)	Improvement	Precision (Without CRR)	Precision (With CRR)	Improvement
Exchange1	71.80%	85.76%	19.44%	10.86%	20.51%	88.93%
Exchange2	70.15%	83.94%	19.66%	10.23%	18.18%	77.69%
Exchange3	73.37%	75.00%	3.43%	5.04%	18.00%	250.42%
Azure1	82.11%	91.25%	11.14%	8.00%	15.38%	90.00%
OneDrive2	75.48%	78.36%	3.69%	37.50%	100%	166.67%
DynamicsCRM1	66.09%	89.64%	35.64%	40.00%	58.33%	45.83%

Table 4: Improvement in Model and Deployment precision with Change-rule refinement(CRR) over Change-rule discovery

7.3 Performance

The suggestion engine is relatively quick, taking approximately 2 seconds to evaluate a pull-request and make a suggestion. In this section, we explore the time it takes to perform the tuning operation across all repositories. This is the most expensive step in the Rex pipeline since it involves multiple runs of the association rule mining algorithm.

Figure 7 shows the tuning time against the size of the repository. Note that both axes are on a logarithmic scale. The largest repository with about 100000 files also takes 370 seconds to tune the model. The two red points in the graph are outliers. They take significantly longer to tune than the other repositories of similar size as the average number of files in each commit is more than other repositories and so each round of association rule mining takes longer.

8 User Study

To understand the relevance of Rex, we performed an extensive user study by sending emails to 328 engineers working across 5 of the repositories on which Rex is deployed. The user study was conducted in three phases:

Phase 1: When we manually examined Rex's suggestions in deployment, we noticed that often, users did not accept some suggestions even though they seemed useful. We therefore asked 30 engineers from 3 teams to subjectively comment on the utility of Rex's suggestions. From their responses, we categorized the suggestions into three categories:

1. Accepted: Some suggestions clearly point out file-changes that are absolutely required and if not acted upon, will lead to bugs/build failures/service disruption. Example 6 in Table 1 shows an example . If the engineer alters a function signature in a script, then they also have to change the way they call the function from another script. Else, service deployment will fail. Engineers usually act on such suggestions. These are what constitute Rex true-positives.

2. Relevant but not accepted: These are suggestions that engineers find useful but do not act upon. For instance, some rules capture the association of test files with core functional code (Example 1 in Table 1). When an engineer makes changes to code by adding a new feature, Rex often recommends editing the test file. While the suggestion makes the engineer consider it, they may not do so, either because the test will delay deployment, or because they decide to add it later. Another example is Example 5 where Rex will make the suggestion, but is unable to infer that the suggestion is valid only if the line-number of the offending code changes. It is up to the engineer to decide if they have indeed changed the line-number of the offending code, and if not, they will not act on the suggestion. However it is still useful since it brought the engineer's attention to the issue.

3. Irrelevant:These suggestions are not relevant to the engineers and thus are not acted upon. For instance, when an engineer is working on a new project, they modify the project configuration file very often, mostly adding new references. Rex therefore learns a lot of spurious change-rules that associate code files with the configuration file. With time, the engineer stops adding reference files, and hence, most suggestions

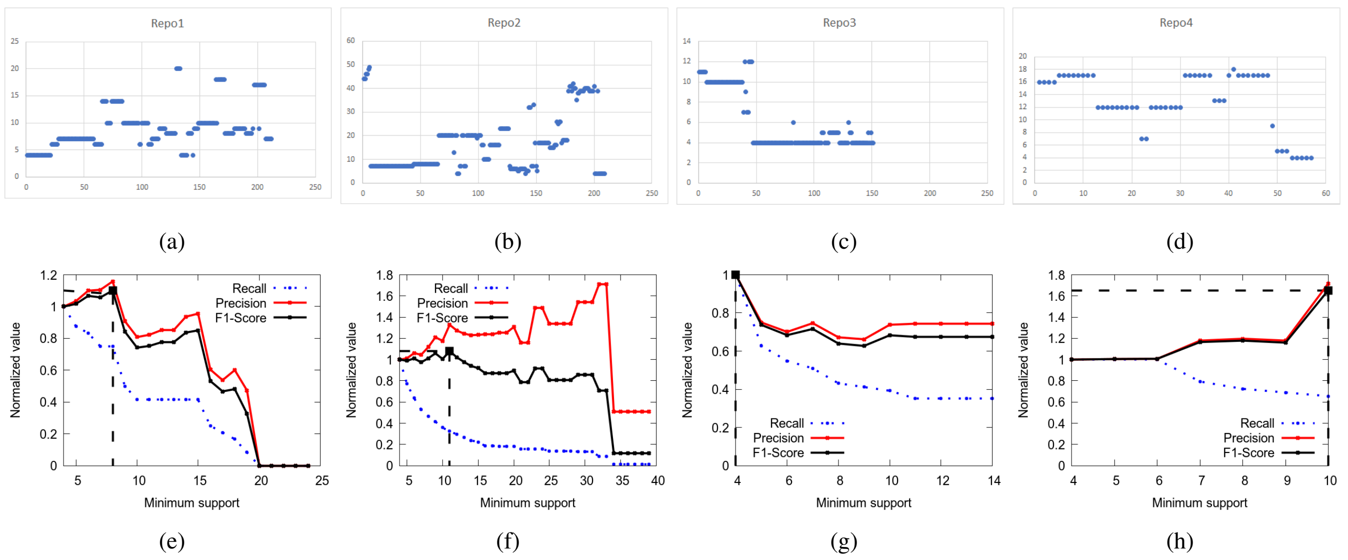


Figure 6: For four repositories, the top row shows how s_{min} varies with time. For each repository, the bottom row shows the precision, recall and F1-score as a function of s_{min}. The black square on the F1-score plot shows the maximum F1-score.

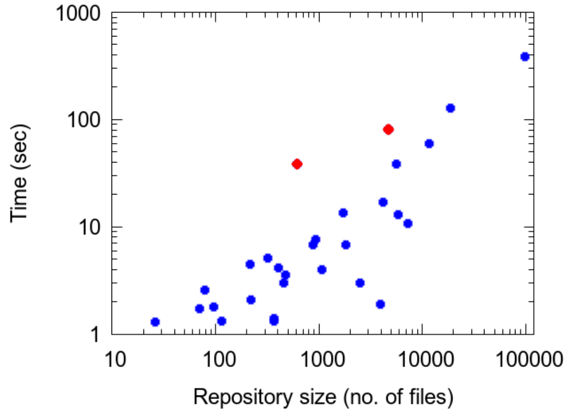


Figure 7: Tuning time vs repository size

based on this change-rule are irrelevant. Since Rex retunes its model regularly, it does eventually drop this change-rule.

Phase 2: Rex true-positives or suggestions that were accepted can be easily estimated by tracking files that got changed in the later iterations of the pull-request. False-positives, on the other hand may either be relevant or irrelevant. To understand what fraction of false-positives may still be relevant, we sent emails to 263 engineers who had not accepted Rex's suggestions. We asked them to categorize Rex's suggestion they saw into one of four categories:

1. The recommendation was relevant but you could not act upon it for some reason.
2. The recommendation was relevant in general but not in this case. Yet, it helped to think through the suggestion.
3. The recommendation was relevant in general but not in this case. You'd rather not have seen the suggestion.
4. The recommendation was not relevant at all.

We received a total of 156 responses. 99 engineers selected the first or second option, i.e., they found the suggestions relevant, i.e. in 99 out of 156 cases, even though Rex's suggestion was a false-positive, it was still useful to the engineer.

Phase 3 : We also used feedback from users to understand the impact of Rex suggestions that were accepted. What if Rex had not made those suggestions and the engineer did not make the correlated changes? To understand this, we sent emails to 65 engineers who had accepted Rex suggestions (true-positives). We provided them with the following options:

1. The recommendation was relevant and had the file not been edited, it could have (broken the build)/(led to service disruption)/(introduced a bug). [High Impact]
2. The recommendation was good to have but the impact of not editing the suggested file would be low. [Low Impact/Good to have]

We received a total of 16 responses. 7 engineers chose option 1. We quote some of their comments here:

"In fact without the suggested change, the code would not have worked."

"If the file is not edited, the build would have failed."

"The suggestion was valid and saved later service disruptions and time"

9 engineers found the suggestion having low impact but without that change, code quality would have been impacted negatively. Some of the responses from engineers include :

"It was good to have edited the additional files for consistency, but it would not have caused any live site impact"

"Even though the build/tests would have been successful, it was a good-to-have suggestion. Adding these files helped unit test the code changes."

From this user study, we infer that Rex is making many more relevant suggestions than the hit rate suggests. Rex is also catching a good number of high-impact suggestions which, if not accepted, would have caused breaks in the build pipeline of the service or even service disruption.

9 Related Work

Rex takes inspiration from two categories of previous work: configuration management in large systems, and code dependency analysis in empirical software engineering. In this section, we describe related-work in these areas.

9.1 Configuration Management

Previous work has explored automated bug and misconfiguration detection using both black-box [16, 24, 30, 31] and white-box techniques [34, 35]. It has also shown how detecting misconfigurations early can help bring down the cost of service disruptions significantly [34].

EnCore [34] uses pre-defined or user-specified templates to detect misconfigurations, which allows it to detect more fine-grained correlations between individual configuration parameters. Through interviews with practitioners, we found that requiring manual inputs posed a severe impediment to adoption. Hence we designed Rex to not require any manual inputs, and to automatically learn templates (generalizations) using change-rule refinement. As a consequence, Rex may not detect rules at as fine a granularity as EnCore.

An orthogonal body of work [14, 25, 28] targets the problem by proposing tool-suites that make it easier for engineers to manage and validate configuration across large services. Facebook's holistic configuration [28] also illustrates the effort required to detect misconfigurations, by using automated canary testing for changed configurations, and using user-defined invariants to drive configuration changes. However, none of these specifically target the problem of correlated configurations explicitly.

9.2 Code Suggestions

Previous work has explored the idea of providing suggestions to engineers to change certain parts of code based on the changes they have already made. Some efforts rely on detecting structural dependencies in code based on program analysis to suggest related components [16, 21, 36]. Others determine couplings between classes in managed code using several semantic and logical techniques [6]. This body of work studies how code dependencies and couplings influence a software engineer's view of related changes. However, they are mostly analyses and learnings, and in most cases, have not been extended to design and deployment of a generic tool that detects such couplings and suggests changes to engineers.

Most related in this space to Rex is work that infers transactions using association rule mining on code version histories [37]. The authors have developed a tool that uses association rule mining to suggest related code changes within an IDE. However, they do not follow it up with inductive generalization/anti-unification which was necessary to reduce the false-recommendations. To speed up the mining process, the consequent of a rule is constrained to have single entity. Hence the rules detected by the tool will be a subset of the rules Rex learns. They mine rules on the fly, each rule taking a few seconds which does not scale well for large-scale deployments. Rex is a more generic technique and has been deployed widely across different services.

MUVI [16] uses frequent itemset mining to find correlated variable accesses in code. If the programmer does not access all correlated variables together, or does not guard them with the same lock, MUVI flags a potential bug. Getafix [23] uses code change analysis to guide testing and to find bugs related to certain properties, such as a missing null-check. Both MUVI and Getafix are designed to discover very specific kinds of bugs, such as multiple access correlations in the case of MUVI and null dereferencing in the case of Getafix. The goal of Rex is to be generic and applicable to a wide range of scenarios across multiple service deployments. We believe that such tools could work very well alongside Rex.

10 Conclusion

This paper presents Rex, a widely deployed and scalable service that performs correlated change analysis using change-rule discovery and change-rule refinement to identify development gaps in code changes being proposed by engineers. Many lessons have been learned during the development and deployment of Rex, which have been outlined and presented in this paper. Most significantly, engineers are always looking for more tools and services to help their process, and Rex fits into their workflow naturally and effectively. Rex has had significant impact in avoiding bad deployments, service outages, build breaks, and buggy commits.

Acknowledgements

We would like to thank developers from Exchange Online, Azure, OneDrive and Dynamics CRM team for providing valuable feedback as part of the user study. We would also like to thank Tom Zimmerman, developers from Azure PIE team, Rana Sulaiman and Robin Schramm for helping us with the initial model and our shepherd Anirudh Sivaraman for providing feedback on the paper.

References

[1] Roslyn: Code syntax analyzer. https://github.com/dotnet/roslyn. [Online; accessed 24-April-2019].

[2] Gregory Piatetsky-Shapiro ,Matthew Mayo. https://www.kdnuggets.com/2016/04/association-rules-apriori-algorithm-tutorial.html. [Online; accessed 24-April-2019].

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[4] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 134–144. IEEE Press, 2015.

[5] R. Bavishi, H. Yoshida, and M. R. Prasad. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 613–624, New York, NY, USA, 2019. ACM.

[6] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. An empirical study on the developers' perception of software coupling. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 692–701, May 2013.

[7] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. *SIGMOD Rec.*, 27(2):85–93, June 1998.

[8] R. Bhagwan, R. Kumar, C. S. Maddila, and A. A. Philip. Orca: Differential bug localization in large-scale services. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 493–509, Carlsbad, CA, 2018. USENIX Association.

[9] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *SIGMOD Rec.*, 26(2):255–264, June 1997.

[10] M. Corporation. Generating diffgrams of xml-files. https://www.nuget.org/packages/XMLDiffPatch/. [Online; accessed 24-April-2019].

[11] GitHub Inc. https://github.com. [Online; accessed 24-April-2019].

[12] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 1–16, New York, NY, USA, 2016. ACM.

[13] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[14] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou. Confvalley: A systematic configuration validation framework for cloud services. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 19:1–19:16, New York, NY, USA, 2015. ACM.

[15] T. Kutsia, J. Levy, and M. Villaret. Anti-unification for unranked terms and hedges. *Journal of Automated Reasoning*, 52(2):155–190, 2014.

[16] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 103–116, New York, NY, USA, 2007. ACM.

[17] Microsoft Azure Cloud Services. https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-choose-me. [Online; accessed 24-April-2019].

[18] Microsoft Azure DevOps. https://azure.microsoft.com/en-in/services/devops/. [Online; accessed 24-April-2019].

[19] Microsoft Azure Documentation. https://docs.microsoft.com/en-in/azure/. [Online; accessed 24-April-2019].

[20] M. Pradel and K. Sen. Deepbugs: A learning approach to name-based bug detection. *CoRR*, abs/1805.11683, 2018.

[21] M. P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 11–20, New York, NY, USA, 2005. ACM.

[22] SalesForce. Rcm for eu18 disruptions of service - aug sept 2018. https://help.salesforce.com/articleView?id=000321150&type=1. [Online; accessed 24-April-2019].

[23] A. Scott, J. Bader, and S. Chandra. Getafix: Learning to fix bugs automatically. *CoRR*, abs/1902.06111, 2019.

[24] A. Scott, J. Bader, and S. Chandra. Getafix: Learning to fix bugs automatically. *CoRR*, abs/1902.06111, 2019.

[25] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein. Acms: The akamai configuration management system. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 245–258, Berkeley, CA, USA, 2005. USENIX Association.

[26] B. W. Silverman. Using kernel density estimates to investigate multimodality. *Journal of the Royal Statistical Society: Series B (Methodological)*, 43(1):97–99, 1981.

[27] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 328–343, New York, NY, USA, 2015. ACM.

[28] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343. ACM, 2015.

[29] The Git Version Control System. https://git-scm.com/.

[30] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, Y.-M. Wang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 17–17, Berkeley, CA, USA, 2004. USENIX Association.

[31] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. *Science of Computer Programming*, 53(2):143–164, 2004.

[32] A. Weiss, A. Guha, and Y. Brun. Tortoise: Interactive system configuration repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 625–636. IEEE, 2017.

[33] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 307–319, New York, NY, USA, 2015. ACM.

[34] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 619–634, Savannah, GA, 2016. USENIX Association.

[35] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *ACM SIGPLAN Notices*, volume 49, pages 687–700. ACM, 2014.

[36] T. Zimmerman, N. Nagappan, K. Herzig, R. Premraj, and L. Williams. An empirical study on the relation between dependency neighborhoods and failures. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 347–356, March 2011.

[37] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

Building An Elastic Query Engine on Disaggregated Storage

Midhul Vuppalapati
Cornell University

Justin Miron
Cornell University

Rachit Agarwal
Cornell University

Dan Truong
Snowflake Computing

Ashish Motivala
Snowflake Computing

Thierry Cruanes
Snowflake Computing

Abstract

We present operational experience running *Snowflake*, a cloud-based data warehousing system with SQL support similar to state-of-the-art databases. Snowflake design is motivated by three goals: (1) compute and storage elasticity; (2) support for multi-tenancy; and, (3) high performance. Over the last few years, Snowflake has grown to serve thousands of customers executing millions of queries on petabytes of data every day.

This paper presents Snowflake design and implementation, along with a discussion on how recent changes in cloud infrastructure (emerging hardware, fine-grained billing, etc.) have altered the many assumptions that guided the design and optimization of Snowflake system. Using data collected from various components of our system during execution of 70 million queries over a 14 day period, our study both deepens the understanding of existing problems and highlights new research challenges along a multitude of dimensions including design of storage systems and high-performance query execution engines.

1 Introduction

Shared-nothing architectures have been the foundation of traditional query execution engines and data warehousing systems. In such architectures, *persistent* data (*e.g.*, customer data stored as tables) is partitioned across a set of compute nodes, each of which is responsible only for its local data. Such shared-nothing architectures have enabled query execution engines that scale well, provide cross-job isolation and good data locality resulting in high performance for a variety of workloads. However, these benefits come at the cost of several major disadvantages:

- **Hardware-workload mismatch:** Shared-nothing architectures make it hard to strike a perfect balance between CPU, memory, storage and bandwidth resources provided by compute nodes, and those required by workloads. For instance, a node configuration that is ideal for bandwidth-intensive compute-light bulk loading may be a poor fit for compute-extensive bandwidth-light complex queries. Many customers, however, want to run a mix of queries without setting up a separate cluster for each query type. Thus, to meet performance goals, resources usually have to be over-provisioned; this results in resource underutilization on an average and in higher operational costs.
- **Lack of Elasticity:** Even if one could match the hardware resources at compute nodes with workload demands, static parallelism and data partitioning inherent to (inelastic) shared-nothing architectures constrain adaptation to data skew and time-varying workloads. For instance, queries run by our customers have extremely skewed intermediate data sizes that vary over five orders of magnitude (§4), and have CPU requirements that change by as much as an order of magnitude within the same hour (§7). Moreover, shared-nothing architectures do not admit efficient elasticity; the usual approach of adding/removing nodes to elastically scale resources requires large amounts of data to be reshuffled. This not only increases network bandwidth requirements but also results in significant performance degradation since the set of nodes participating in data reshuffling are also responsible for query processing.

Traditional data warehousing systems were designed to operate on recurring queries on data with predictable volume and rate, *e.g.*, data coming from within the organization: transactional systems, enterprise resource planning application, customer relationship management applications, etc. The situation has changed significantly. Today, an increasingly large fraction of data comes from less controllable, external sources (*e.g.*, application logs, social media, web applications, mobile systems, etc.) resulting in ad-hoc, time-varying, and unpredictable query workloads. For such workloads, shared-nothing architectures beget high cost, inflexibility, poor performance and inefficiency, which hurts production applications and cluster deployments.

To overcome these limitations, we designed Snowflake — an elastic, transactional query execution engine with SQL support comparable to state-of-the-art databases. The key insight in Snowflake design is that the aforementioned limitations of shared-nothing architectures are rooted in tight coupling of compute and storage, and the solution is to decouple the two! Snowflake thus disaggregates compute from persistent storage; customer data is stored in a persistent data store (*e.g.*, Amazon S3 [5], Azure Blob Storage [8], etc.) that provides high availability and on-demand elasticity. Compute elasticity is achieved using a pool of pre-warmed nodes, that can be assigned to customers on an on-demand basis.

Snowflake system design uses two key ideas (§2). First, it uses a custom-designed storage system for management and exchange of ephemeral/intermediate data that is exchanged between compute nodes during query execution (*e.g.*, tables exchanged during joins). Such an ephemeral storage system was necessary because existing persistent data stores [5, 8] have two main limitations: (1) they fall short of providing the necessary latency and throughput performance to avoid compute tasks being blocks on exchange of intermediate data; and (2) they provide stronger availability and durability semantics than what is needed for intermediate data. Second, Snowflake uses its ephemeral storage system not only for intermediate data, but also as a write-through "cache" for persistent data. Combined with a custom-designed query scheduling mechanism for disaggregated storage, Snowflake is able to reduce the additional network load caused by compute-storage disaggregation as well as alleviate the performance overheads of reduced data locality.

Snowflake system has now been active for several years and today, serves thousands of customers executing millions of queries over petabytes of data, on a daily basis. This paper describes Snowflake system design, with a particular focus on ephemeral storage system design, query scheduling, elasticity and efficiently supporting multi-tenancy. We also use statistics collected during execution of $\sim$70 million queries over a period of 14 contiguous days in February 2018 to present a detailed study of network, compute and storage characteristics in Snowflake. Our key findings are:

- Customers submit a wide variety of query types; for example, read-only queries, write-only queries and read-write queries, each of which contribute to $\sim$28%, $\sim$13% and $\sim$59%, respectively, of all customer queries.
- Intermediate data sizes can vary over multiple orders of magnitude across queries, with some queries exchanging hundreds of gigabytes or even terabytes of intermediate data. The amount of intermediate data generated by a query has little or no correlation with the amount of persistent data read by the query or the execution time of the query.
- Even with a small amount of local storage capacity, skewed access distributions and temporal access patterns common in data warehouses enable reasonably high average cache hit rates (60-80% depending on the type of query) for persistent data accesses.
- Several of our customers exploit our support for elasticity (for $\sim$20% of the clusters). For cases where customers do request elastic scaling of resources, the number of compute nodes in their cluster can change by as much as two orders of magnitude during the lifetime of the cluster.
- While the peak resource utilization can be high, the average resource utilization is usually low. We observe average CPU, Memory, Network Tx and Network Rx utilizations of $\sim$51%, $\sim$19%, $\sim$11%, $\sim$32%, respectively.

Our study both corroborates exciting ongoing research directions in the community, as well as highlights several interesting venues for future research:

- **Decoupling of compute and ephemeral storage:** Snowflake decouples compute from persistent storage to achieve elasticity. However, currently, compute and ephemeral storage are still tightly coupled. As we show in §4, the ratio of compute capacity and ephemeral storage capacity in our production clusters can vary by several orders of magnitude, leading to either under utilization of CPU or thrashing of ephemeral storage, for ad-hoc query processing workloads. To that end, recent academic work on decoupling compute from ephemeral storage [22, 27] is of extreme interest. However, more work is needed in ephemeral storage system design, especially in terms of providing fine-grained elasticity, multi-tenancy, and cross-query isolation (§4, §7).
- **Deep storage hierarchy:** Snowflake ephemeral storage system, similar to recent work on compute-storage disaggregation [14, 15], uses caching of frequently read persistent data to both reduce the network traffic and to improve data locality. However, existing mechanisms for improving caching and data locality were designed for two-tier storage systems (memory as the main tier and HDD/SSD as the second tier). As we discuss in §4, the storage hierarchy in our production clusters is getting increasingly deeper, and new mechanisms are needed that can efficiently exploit the emerging deep storage hierarchy.
- **Pricing at sub-second timescales:** Snowflake achieves compute elasticity at fine-grained timescales by serving customers using a pool of pre-warmed nodes. This was cost-efficient with cloud pricing at hourly granularity. However, most cloud providers have recently transitioned to sub-second pricing [6], leading to new technical challenges in efficiently achieving resource elasticity and resource sharing across multiple tenants. Resolving these challenges may require design decisions and tradeoffs that may be different from those in Snowflake's current design (§7).

This paper focuses on Snowflake system architecture along with compute, storage and network characteristics observed in our production clusters. Accordingly, we focus on details that are necessary to make the paper self-contained (§2). For details on Snowflake query planning, optimization, concurrency control mechanisms, etc., please refer to [12]. To aid future research and studies, we are releasing an anonymized version of the dataset used in this paper; this dataset comprising statistics collected per-query for ∼70 million queries. The dataset is available publicly along with documentation and scripts to reproduce all results in this paper at https://github.com/resource-disaggregation/snowset.

Our study has an important caveat. It focuses on a specific system (Snowflake), a specific workload (SQL queries), and a specific cloud infrastructure (S3). While our system is large-scale, has thousands of customers executing millions of queries, and runs on top of one of the most prominent infrastructures, it is nevertheless limited. We leave it to future work an evaluation of whether our study and observations generalize to other systems, workloads and infrastructures. However, we are hopeful that just like prior workload studies on network traffic characteristics [9] and cloud workloads [28] (each of which also focused on a specific system implementation running a specific workload on a specific infrastructure) fueled and aided research in the past, our study and publicly released data will be useful for the community.

2 Design Overview

We provide an overview of Snowflake design. Snowflake treats persistent and intermediate data differently; we describe these in §2.1, followed by a high-level overview of Snowflake architecture (§2.2) and query execution process (§2.3).

2.1 Persistent and Intermediate data

Like most query execution engines and data warehousing systems, Snowflake has three forms of application state:

- *Persistent data* is customer data stored as tables in the database. Each table may be read by many queries, over time or even concurrently. These tables are thus long-lived and require strong durability and availability guarantees.
- *Intermediate data* is generated by query operators (*e.g.*, joins) and is usually consumed by nodes participating in executing that query. Intermediate data is thus short-lived. Moreover, to avoid nodes being blocked on intermediate data access, low-latency high-throughput access to intermediate data is preferred over strong durability guarantees. Indeed, in case of failures happening during the (short) lifetime of intermediate data, one can simply rerun the part of the query that produced it.
- *Metadata* such as object catalogs, mapping from database tables to corresponding files in persistent storage, statistics, transaction logs, locks, etc.

This paper primarily focuses on persistent and intermediate data, as the volume of metadata is typically relatively small and does not introduce interesting systems challenges.

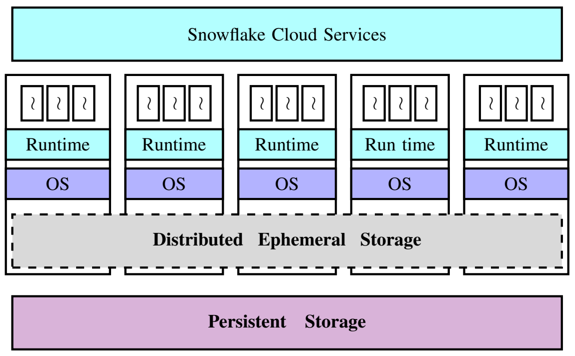


Figure 1: **Snowflake (Virtual) Warehouse Architecture (§2.2).**

2.2 End-to-end System Architecture

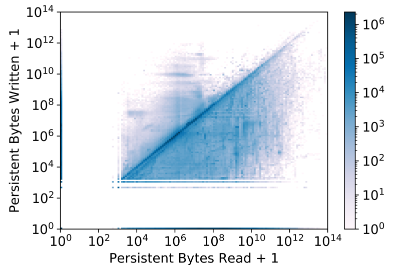
Figure 1 shows the high-level architecture for Snowflake. It has four main components — a centralized service for orchestrating end-to-end query execution, a compute layer, a distributed ephemeral storage system and a persistent data store. We describe each of these below[1].

Centralized Control via Cloud Services. All Snowflake customers interact with and submit queries to a centralized layer called Cloud Services (CS) [12]. This layer is responsible for access control, query optimization and planning, scheduling, transaction management, concurrency control, etc. CS is designed and implemented as a multi-tenant and long-lived service with sufficient replication for high availability and scalability. Thus, failure of individual service nodes does not cause loss of state or availability, though some of the queries may fail and be re-executed transparently.

Elastic Compute via Virtual Warehouse abstraction. Customers are given access to computational resources in Snowflake through the abstraction of a *Virtual Warehouse* (VW). Each VW is essentially a set of AWS EC2 instances on top which customer queries execute in a distributed fashion. Customers pay for compute-time based on the VW size. Each VW can be elastically scaled on an on-demand basis upon customer request. To support elasticity at fine-grained timescales (*e.g.*, tens of seconds), Snowflake maintains a pool of pre-warmed EC2 instances; upon receiving a request, we simply add/remove EC2 instances to/from that VW (in case of addition, we are able to support most requests directly from our pool of pre-warmed instances thus avoiding instance startup time). Each VW may run multiple concurrent queries. In fact, many of our customers run multiple VWs (*e.g.*, one for data ingestion, and one for executing OLAP queries).

Elastic Local Ephemeral Storage. Intermediate data has different performance requirements compared to persistent data (§2.1). Unfortunately, existing persistent data stores do not meet these requirements (*e.g.*, S3 does not provide the desired low-latency and high-throughput properties needed

[1]This paper describes design and implementation of Snowflake using Amazon Web Services as an example infrastructure; however, Snowflake runs on Microsoft Azure and Google Cloud Platform as well.



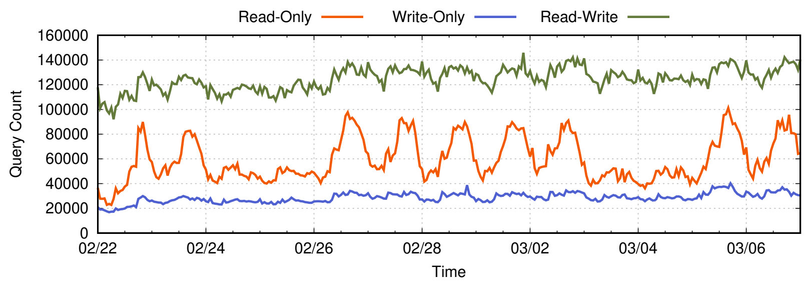


Figure 2: **Persistent data read/write, and submission time characteristics of queries in our dataset. (left)** Scatter plot with each point representing a query based on the total number of persistent data bytes read and written by the query. The density of points is concentrated along three regions: (1) read-only queries along x-axis; (2)write-only queries along y-axis; and, (3) read-write queries along the middle region. **(right)** for each query class, the number of queries submitted at different times of day over the 14 day period, binned on an hourly basis. Read-only queries have significantly higher variation in load compared to the other query classes, with spikes during daytime hours on weekdays.

for intermediate data to ensure minimal blocking of compute nodes); hence, we built a distributed ephemeral storage system custom-designed to meet the requirements of intermediate data in our system. The system is co-located with compute nodes in VWs, and is explicitly designed to automatically scale as nodes are added or removed. We provide more details in §4 and §6, but note here that as nodes are added and removed, our ephemeral storage system does not require data repartitioning or reshuffling (thus alleviating one of the core limitations of shared-nothing architectures). Each VW runs its own independent distributed ephemeral storage system which is used only by queries running on that particular VW.

Elastic Remote Persistent Storage. Snowflake stores all its persistent data in a remote, disaggregated, persistent data store. We store persistent data in S3 despite the relatively modest latency and throughput performance because of S3's elasticity, high availability and durability properties. S3 supports storing immutable files — files can only be overwritten in full and do not even allow append operations. However, S3 supports read requests for parts of a file. To store tables in S3, Snowflake partitions them horizontally into large, immutable files that are equivalent to blocks in traditional database systems [12]. Within each file, the values of each individual attribute or column are grouped together and compressed, as in PAX [2]. Each file has a header that stores offset of each column within the file, enabling us to use the partial read functionality of S3 to only read columns that are needed for query execution.

All VWs belonging to the same customer have access to the same shared tables via remote persistent store, and hence do not need to physically copy data from one VW to another.

2.3 End-to-end query execution

Query execution begins with customers submitting their query text to CS for execution on a specific customer VW. At this stage, CS performs query parsing, query planning and optimization, producing a set of tasks that need to be executed. It then schedules these tasks on compute nodes of the VW; each task may perform read/write operations on both ephemeral storage system and remote persistent data store. We describe the scheduling and query execution mechanisms in Snowflake in §5. CS continually tracks the progress of each query, collects performance counters, and upon detecting a node failure, reschedules the query on compute nodes within the VW. Once the query is executed, the corresponding result is returned back to the CS and eventually to the customer.

3 Dataset

Snowflake collects statistics at each layer of the system — CS collects and stores information for each individual VW (size over time, instance types, failure statistics, etc.), performance counters for individual queries, time spent in different phases of query execution, etc. Each node collects statistics for ephemeral and persistent store accesses, resource (CPU, memory and bandwidth) utilization characteristics, compression properties, etc. To aid future research and studies, we are publicly releasing a dataset containing most of these statistics for ∼70 million queries over a period of 14 days, aggregated per-query. The dataset is publicly available at https://github.com/resource-disaggregation/snowset. For privacy reasons, the dataset does not contain information on query plans, table schemas and per-file access frequencies. To ensure reproducibility, this paper uses only those statistics that are contained in publicly released dataset.

Query Classification. We classify queries in the dataset based on number of persistent data bytes read and written (Figure 2 (left)). Figure 2 (right) shows number of queries submitted at different times of day for each query class.

- **Read-only queries:** Queries along the x-axis are the ones that do not write any persistent data; however, the amount of data read by these queries can vary over nine orders of magnitude. These queries contribute to ∼28% of all customer queries, and represent ad-hoc and interactive OLAP

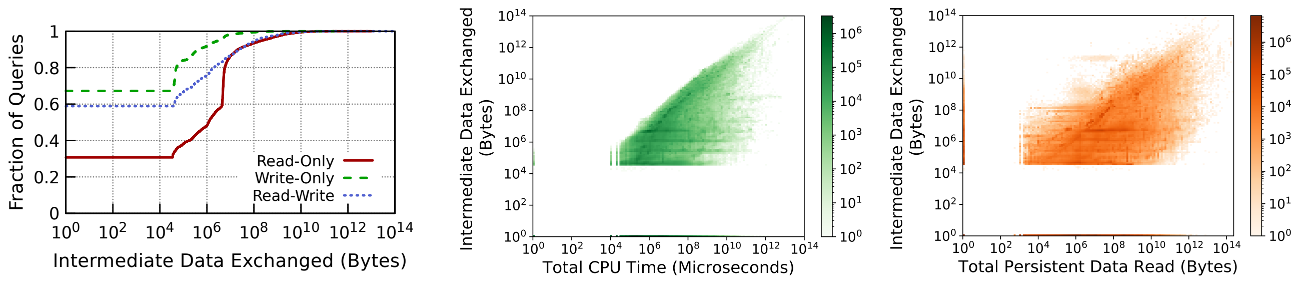


Figure 3: **Intermediate data characteristics. (left)** Intermediate data sizes vary over multiple orders of magnitude across queries; a non-trivial fraction of queries in each query class exchange zero intermediate data, while some read-only and read-write queries exchange $10-100$TB of intermediate data. **(center)** Scatter plot with each point representing a query based on its total CPU time and amount of intermediate data exchanged by the corresponding query; queries with the same total CPU time exchange vastly different amounts of intermediate data. **(right)** Scatter plot with each point representing a query based on its total persistent data read and amount of intermediate data exchanged by the corresponding query; queries that read the same amount of persistent data exchange vastly different amounts of intermediate data.

workloads [10] typical in data warehouses, where the result is usually small in size and is directly returned to the client. The right figure demonstrates an interesting trend for read-only queries — the number of such queries submitted by customers spike during daytime hours on weekdays.

- **Write-only queries:** Queries along the y-axis are the ones that do not read any persistent data; however, the amount of data written by these queries can vary over eight orders of magnitude. These queries contribute to ~13% of all customer queries, and essentially represent data ingestion queries that bring data into the system. Unlike read-only queries, we observe that the rate of submission for write-only queries is fairly consistent across time.
- **Read-Write (RW) queries:** The region in the middle of the plot contains ~59% of customer queries, and represents queries that both read and write persistent data. Here we see a wide spectrum of queries. Many queries have read-write ratio close to 1, in terms of number of bytes, that represent Extract Transform Load (ETL) pipelines [29, 31, 32] typical in data warehouses. For other queries, the read-write ratio can vary over multiple orders of magnitude.

The above classification is based on number of persistent data bytes read and written by the queries, a measure that is not an artifact of Snowflake's architecture; rather, it is a property of the queries themselves. Indeed, even if these queries were to run on, say, any other data analytics framework (*e.g.*, Hadoop) or even a single node database (*e.g.*, MySQL), the persistent read/write characteristics would remain the same. We do not classify queries based on semantics as the focus of this paper is on systems characteristics, and our dataset does not contain detailed information about individual query plans. We will use the above query classification throughout the paper.

4 Ephemeral Storage System

Snowflake uses a custom-designed distributed storage system for management and exchange of intermediate data, due to two limitations in existing persistent data stores [5, 8]. First, they fall short of providing the necessary latency and throughput performance to avoid compute tasks being blocks on intermediate data exchange. Second, they provide much stronger availability and durability semantics than what is needed for intermediate data. Our ephemeral storage system allows us to overcome both these limitations. Tasks executing query operations (*e.g.*, joins) on a given compute node write intermediate data locally; and, tasks consuming the intermediate data read it either locally or remotely over the network (depending on the node where the task is scheduled, §5).

4.1 Storage Architecture, and Provisioning

We made two important design decisions in our ephemeral storage system. First, rather than designing a pure in-memory storage system, we decided to use both memory and local SSDs — tasks write as much intermediate data as possible to their local memory; when memory is full, intermediate data is spilled to local SSDs. Our rationale is that while purely in-memory systems can achieve superior performance when entire data fits in memory, they are too restrictive to handle the variety of our target workloads. Figure 3 (left) shows that there are queries that exchange hundreds of gigabytes or even terabytes of intermediate data; for such queries, it is hard to fit all intermediate data in main memory.

The second design decision was to allow intermediate data to spill into remote persistent data store in case the local SSD capacity is exhausted. Spilling intermediate data to S3, instead of other compute nodes, is preferable for a number of reasons — it does not require keeping track of intermediate data location, it alleviates the need for explicitly handling out-of-memory or out-of-disk errors for large queries, and overall, allows to keep our ephemeral storage system thin and highly performant.

Future Directions. For performance-critical queries, we want intermediate data to entirely fit in memory, or at least in SSDs,

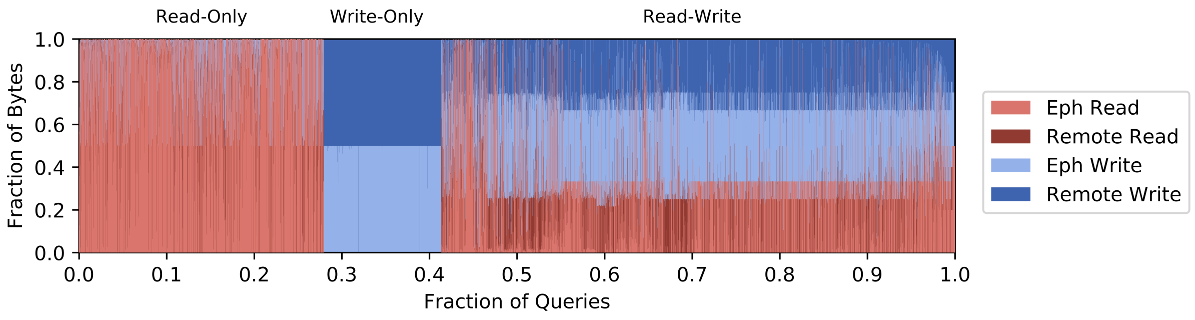


Figure 4: **Persistent data I/O traffic distribution between ephemeral storage system and remote persistent data store.** Each vertical bar corresponds to a query, the four colors correspond to read/write from ephemeral/remote systems, and the y-axis represents the fraction of total persistent data that was served by the corresponding storage system. Queries are sorted, within each class, in decreasing order of number of persistent data bytes read/written. Discussion in §4.2.

and not spill to S3. This requires accurate resource provisioning. However, provisioning CPU, memory and storage resources while achieving high utilization turns out to be challenging due to two reasons. The first reason is limited number of available node instances (each providing a fixed amount of CPU, memory and storage resources), and significantly more diverse resource demands across queries. For instance, Figure 3 (center) shows that, across queries, the ratio of compute requirements and intermediate data sizes can vary by as much as six orders of magnitude. The available node instances simply do not provide enough options to accurately match node hardware resources with such diverse query demands.

Second, even if we could match node hardware resources with query demands, accurately provisioning memory and storage resources requires a priori knowledge of intermediate data size generated by the query. However, our experience is that predicting the volume of intermediate data generated by a query is hard, or even impossible, for most queries. As shown in Figure 3, intermediate data sizes not only vary over multiple orders of magnitude across queries, but also have little or no correlation with amount of persistent data read or the expected execution time of the query.

To resolve the first challenge, we could decouple compute from ephemeral storage. This would allow us to match available node resources with query resource demands by independently provisioning individual resources. However, the challenge of unpredictable intermediate data sizes is harder to resolve. For such queries, simultaneously achieving high performance and high resource utilization would require both decoupling of compute and ephemeral storage, as well as efficient techniques for fine-grained elasticity of ephemeral storage system. We discuss the latter in more detail in §6.

4.2 Persistent Data Caching

One of the key observations we made during early phases of ephemeral storage system design is that intermediate data is short-lived. Thus, while storing intermediate data requires large memory and storage capacity *at peak*, the demand is low *on an average*. This allows statistical multiplexing of our ephemeral storage system capacity between intermediate data and frequently accessed persistent data. This improves performance since (1) queries in data warehouse systems exhibit highly skewed access patterns over persistent data [10]; and (2) ephemeral storage system performance is significantly better than that of (existing) remote persistent data stores.

Snowflake enables statistical multiplexing of ephemeral storage system capacity between intermediate data and persistent data by "opportunistically" caching frequently accessed persistent data files, where opportunistically refers to the fact that intermediate data storage is always prioritized over caching persistent data files. However, a persistent data file cannot be cached on any node — Snowflake assigns input file sets for the customer to nodes using consistent hashing over persistent data file names. A file can only be cached at the node to which it consistently hashes to; each node uses a simple LRU policy to decide caching and eviction of persistent data files. Given the performance gap between our ephemeral storage system and remote persistent data store, such opportunistic caching of persistent data files improves the execution time for many queries in Snowflake. Furthermore, since storage of intermediate data is always prioritized over caching of persistent data files, such an opportunistic performance improvement in query execution time can be achieved without impacting performance for intermediate data access.

Maintaining the right system semantics during opportunistic caching of persistent data files requires a careful design. First, to ensure data consistency, the "view" of persistent files in ephemeral storage system must be consistent with those in remote persistent data store. We achieve this by forcing the ephemeral storage system to act as a write-through cache for persistent data files. Second, consistent hashing of persistent data files on nodes in a naïve way requires reshuffling of cached data when VWs are elastically scaled. We implement a lazy consistent hashing optimization in our ephemeral storage system that avoids such data reshuffling *altogether*; we describe this when we discuss Snowflake elasticity in §6.

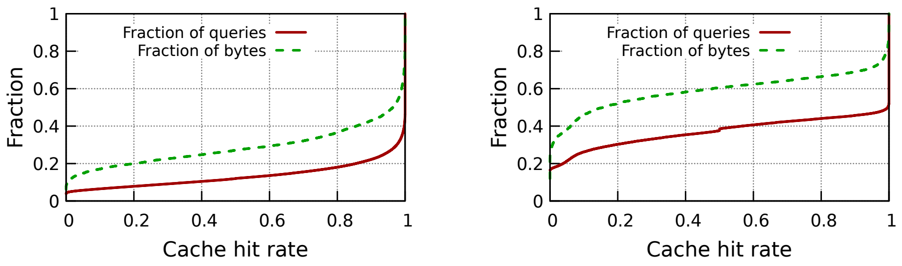


Figure 5: **Cache hit rate distribution of Read-only (left) and Read-Write queries (right).** We plot the CDFs for both, the queries (independent of their persistent data read sizes) and for bytes (queries weighed by the amount of persistent data read). For example, for Read-only queries, 80% of the queries have hit rate greater than 75%, but these queries account for only a little more than 60% of the total number of persistent bytes read by all Read-only queries.

Persistent data being opportunistically cached in the ephemeral storage system means that some subset of persistent data access requests could be served by the ephemeral storage system (depending on whether or not there is a cache hit). Figure 4 shows the persistent data I/O traffic distribution, in terms of fraction of bytes, between the ephemeral storage system and remote persistent data store. The write-through nature of our ephemeral storage system results in amount of data written to ephemeral storage being roughly of the same magnitude as the amount of data written to remote persistent data store (they are not always equal because of prioritizing storage of intermediate data over caching of persistent data).

Even though our ephemeral storage capacity is significantly lower than that of a customer's persistent data (around 0.1% on an average), skewed file access distributions and temporal file access patterns common in data warehouses [7] enable reasonably high cache hit rates (avg. hit rate is close to 80% for read-only queries and around 60% for read-write queries). Figure 5 shows the hit rate distributions across queries. The median hit rates are even higher.

Future Directions. Figure 4 and Figure 5 suggest that more work is needed on caching. In addition to locality of reference in access patterns, cache hit rate also depends on effective cache size available to the query relative to the amount of persistent data accessed by the query. The effective cache size, in turn, depends on both the VW size and the volume of intermediate data generated by concurrently executing queries. Our preliminary analysis has not led to any conclusive observations on the impact of the above two factors on the observed cache hit rates, and a more fine-grained analysis is needed to understand factors that impact cache hit rates.

We highlight two additional technical problems. First, since end-to-end query performance depends on both, cache hit rate for persistent data files and I/O throughput for intermediate data, it is important to optimize how the ephemeral storage system splits capacity between the two. Although we currently use the simple policy of always prioritizing intermediate data, it may not be the optimal policy with respect to end-to-end performance objectives (*e.g.*, average query completion time across all queries from the same customer). For example, it may be better to prioritize caching a persistent data file that is going to be accessed by many queries over intermediate data that is accessed by only one. It would be interesting to explore extensions to known caching mechanisms that optimize for end-to-end query performance objectives [7] to take intermediate data into account.

Second, existing caching mechanisms were designed for two-tier storage systems (memory as the main tier and HDD/SSD as the second tier). In Snowflake, we already have three tiers of hierarchy with compute-local memory, ephemeral storage system and remote persistent data store; as emerging non-volatile memory devices are deployed in the cloud and as recent designs on remote ephemeral storage systems mature [22], the storage hierarchy in the cloud will get increasingly deeper. Snowflake uses traditional two-tier mechanisms — each node implements a local LRU policy for evictions from local memory to local SSD, and an independent LRU policy for evictions from local SSD to remote persistent data store. However, to efficiently exploit the deepening storage hierarchy, we need new caching mechanisms that can efficiently coordinate caching across multiple tiers.

We believe many of the above technical challenges are not specific to Snowflake, and would apply more broadly to any distributed application built on top of disaggregated storage.

5 Query (Task) Scheduling

We now describe the query execution process in Snowflake. Customers submit their queries to the Cloud Services (CS) for execution on a specific VW. CS performs query parsing, query planning and optimization, and creates a set of tasks to be scheduled on compute nodes of the VW.

Locality-aware task scheduling. To fully exploit the ephemeral storage system, Snowflake colocates each task with persistent data files that it operates on using a locality-aware scheduling mechanism (recall, these files may be cached in ephemeral storage system). Specifically, recall that Snowflake assigns persistent data files to compute nodes using consistent

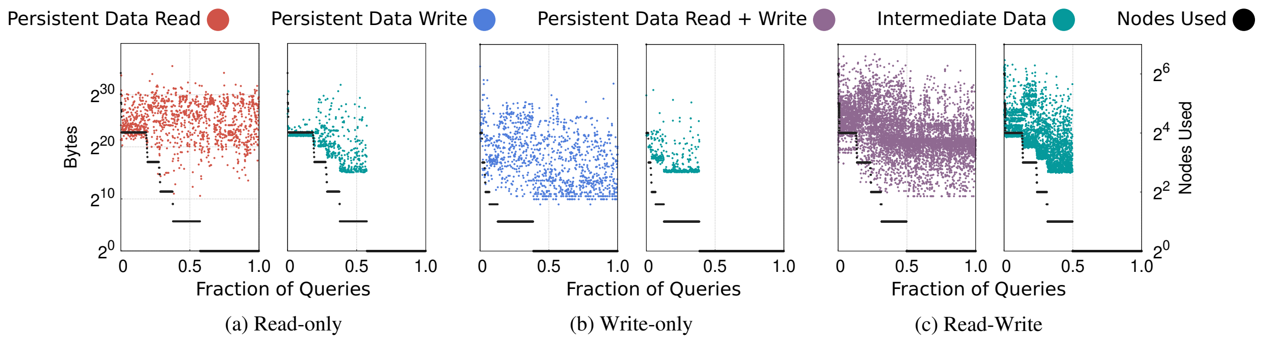


Figure 6: **Persistent data read / write and intermediate data exchange characteristics of queries sorted by the number of nodes used.** Each plot uses the same axis for bytes (left axis) and nodes used (right axis). Persistent Read and write bytes vary by three orders of magnitude across each node count.

hashing over table file names. Thus, for a fixed VW size, each persistent data file is cached on a specific node. Snowflake schedules the task that operates on a persistent data file to the node on which its file consistently hashes to.

As a result of this scheduling scheme, query parallelism is tightly coupled with consistent hashing of files on nodes — a query is scheduled for cache locality and may be distributed across all the nodes in the VW. For instance, consider a customer that has 1million files worth of persistent data, and is running a VW with 10 nodes. Consider two queries, where the first query operates on 100 files, and the second query operates on 100,000 files; then, with high likelihood, both queries will run on all the 10 nodes because of files being consistently hashed on to all the 10 nodes.

Figure 6 illustrates this— the number of persistent bytes read and written vary over orders of magnitude, almost independent of the number of nodes in the VW. As expected, the intermediate data exchanged over the network increases with the number of nodes used.

Work stealing. It is known that consistent hashing can lead to imbalanced partitions [19]. In order to avoid overloading of nodes and improve load balance, Snowflake uses *work stealing*, a simple optimization that allows a node to steal a task from another node if the expected completion time of the task (sum of execution time and waiting time) is lower at the new node. When such work stealing occurs, the persistent data files needed to execute the task are read from remote persistent data store rather than the node at which the task was originally scheduled on. This avoids increasing load on an already overloaded node where the task was originally scheduled (note that work stealing happens only when a node is overloaded).

Future Directions. Schedulers can place tasks onto nodes using two extreme options: one is to colocate tasks with their cached persistent data, as in our current implementation. As discussed in the example above, this may end up scheduling all queries on all nodes in the VW; while such a scheduling policy minimizes network traffic for reading persistent data, it may lead to increased network traffic for intermediate data exchange. The other extreme is to place all tasks on a single node. This would obviate the need of network transfers for intermediate data exchange but would increase network traffic for persistent data reads. Neither of these extremes may be the right choice for all queries. It would be interesting to codesign query schedulers that would pick just the right set of nodes to obtain a sweet spot between the two extremes, and then schedule individual tasks onto these nodes.

6 Resource Elasticity

In this section, we discuss how BlowFish design achieves one of its core goals: resource elasticity, that is, scaling of compute and storage resources on an on-demand basis.

Disaggregating compute from persistent storage enables Snowflake to independently scale compute and persistent storage resources. Storage elasticity is offloaded to persistent data stores [5]; compute elasticity, on the other hand, is achieved using a pre-warmed pool of nodes that can be added/removed to/from customer VWs on an on-demand basis. By keeping a pre-warmed pool of nodes, Snowflake is able to provide compute elasticity at the granularity of tens of seconds.

6.1 Lazy Consistent Hashing

One of the challenges that Snowflake had to resolve in order to achieve elasticity efficiently is related to data management in ephemeral storage system. Recall that our ephemeral storage system opportunistically caches persistent data files; each file can be cached only on the node to which it consistently hashes to within the VW. The problem is similar to shared-nothing architectures: any fixed partitioning mechanism (in our case, consistent hashing) requires large amounts of data to be reshuffled upon scaling of nodes; moreover, since the very same set of nodes are also responsible for query processing, the system observes a significant performance impact during the scaling process.

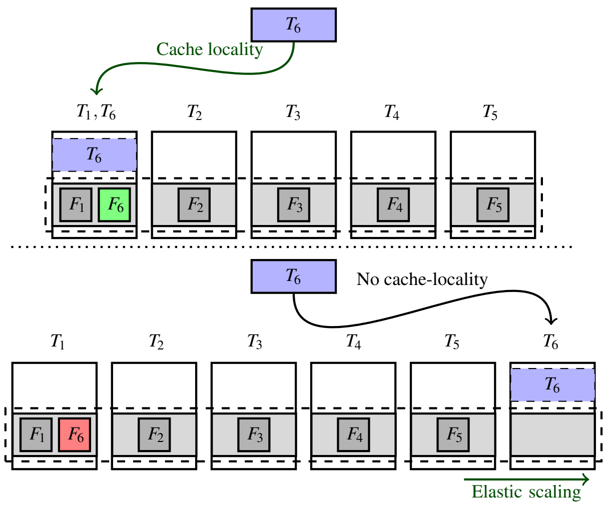


Figure 7: **Snowflake uses lazy consistent hashing to avoid data reshuffling during elastic scaling of VWs. (Top)** VW in steady state with all task inputs cached. **(Bottom)** VW immediately after adding one node. See discussion in §6.1.

Snowflake resolves this challenge using a lazy consistent hashing mechanism, that completely avoids any reshuffling of data upon elastic scaling of nodes by exploiting the fact that a copy of cached data is stored at remote persistent data store. Specifically, Snowflake relies on the caching mechanism to eventually "converge" to the right state. For instance, consider the example in Figure 7 that shows a VW with 6 tasks $T_1, T_2, \ldots, T_6$, with task T_i operating on a single file F_i. Suppose at time t_0, we have 5 nodes in the VW and that node N_1 stores files F_1 and F_6, and nodes $N_2 - N_5$ store file $F_2 - F_5$, respectively. Suppose at time $t > t_0$, a node N_6 is added to the warehouse. Then, rather than immediately reshuffling the files (which would result in F_6 being moved from node N_1 to N_6), Snowflake will wait until task T_6 is executed again. When the next time T_6 is scheduled (*e.g.*, due to work stealing or the same query being executed again), Snowflake will schedule it on N_6 since consistent hashing will now place file F_6 on that node. At this time, file F_6 will be read by N_6 from remote persistent store and cached locally. File F_6 on node N_1 will no longer be accessed and will eventually be evicted from the cache. Such lazy caching allows Snowflake to achieve locality without reshuffling data and without interfering with ongoing queries on nodes already in the VW.

6.2 Elasticity Characteristics

Our customer warehouses exhibit several interesting elasticity characteristics. Figure 8 shows that many of our customers already exploit our support for elasticity (for ∼20% of the VW). For such cases where customers do request VW resizing, the number of nodes in their VW can change by as much as two orders of magnitude during the lifetime of the VW! Figure 9 (top) shows two cases where customers leverage elasticity rather aggressively (even at hourly granularity).

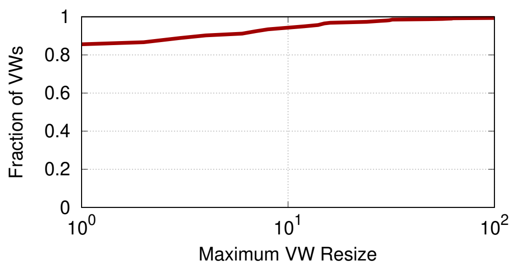


Figure 8: **VW elasticity usage.** 82% of all VW never exploit elasticity; however, some of the customers elastically scale their VW size by as much as two orders of magnitude!

Future Directions. Figure 8 also shows that our customers do not exploit our support for elasticity for more than 80% of the VW. Even for customers that do request VW resizing, there are opportunities for further optimizations — Figure 9 shows that query inter-arrival times in these VW are much finer-grained than the granularity of VW scaling requested by our customers. We believe the main reason for these characteristics is that customers lack the visibility (and the right demand estimation) into the system to accurately request scaling their VW to meet the demand. VW1 in Figure 9, for instance, is one of the heavily utilized VW from a large customer; the characteristics for this VW demonstrate how elastic scaling can mismatch the demand (approximated by query inter-arrival time). Since the time the dataset in the paper was recorded, a lot of work has been done to improve support for auto-scaling VW at the granularity of inter-query arrival times. However, much more work needs to be done.

First, we would like to achieve elasticity at intra-query granularity. Specifically, resource consumption can vary significantly even within the lifetime of individual queries. This is particularly prevalent in long running queries with many internal stages. Hence, in addition to auto-scaling VW at the granularity of query inter-arrivals, we would ideally like to support some level of task-level elasticity even during the execution of a query.

Second, we would like to explore serverless-like platforms. Serverless infrastructures such as AWS Lambda, Azure Functions and Google Cloud Functions which provide auto-scaling, high elasticity and fine-grained billing, are seeing increasing adoption across many application types. However, the key barrier for Snowflake to transition to existing serverless infrastructures is their lack of support for isolation, both in terms of security and performance [34]. Snowflake serves several customers who store and query sensitive and confidential data, and thus require strong isolation guarantees. One possibility for Snowflake is to build its own custom serverless-like compute platform. We believe this is an intriguing direction to explore, but will require resolving several challenges in efficient remote ephemeral storage access (§4.1), and in multi-tenant resource sharing (which we will discuss in §7).

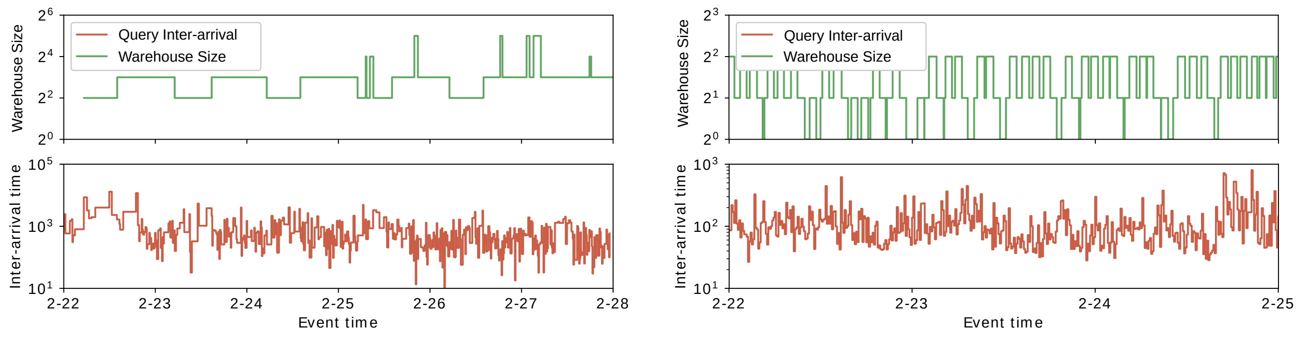


Figure 9: **Comparison of VW resizing (elastic scaling) and query inter-arrival times, binned per minute, for two heavily utilized VW1 (left) and VW2 (right).** Customers request VW resizing at much coarser grained timescales than query inter-arrival times for both VW. Note the difference in x-axis: 7 days and 4 days for VW1 and VW2, respectively.

7 Multi-tenancy

Snowflake currently supports multi-tenancy through the VW abstraction. Each VW operates on an isolated set of nodes, with its own ephemeral storage system. This allows Snowflake to provide performance isolation to its customers. In this section, we present a few system-wide characteristics for our VWs and use these to motivate an alternate sharing based architecture for Snowflake.

The VW architecture in Snowflake leads to the traditional performance isolation versus utilization tradeoff. Figure 10 (top four) show that our VWs achieve fairly good, but not ideal, average CPU utilization; however, other resources are usually underutilized on an average. Figure 11 provides some reasons for the low average resource utilization in Figure 10 (top four): the figure shows the variability of resource usage across VW; specifically, we observe that for up to 30% of VW, standard deviation of CPU usage over time is as large as the mean itself. This results in underutilization as customers tend to provision VWs to meet peak demand. In terms of peak utilization, several of our VWs experience periods of heavy utilization, but such high-utilization periods are not necessarily synchronized across VWs. An example of this is shown in Figure 10 (bottom two), where we see that over a period of two hours, there are several points when one VW's utilization is high while the other VW's utilization is simultaneously low.

While we were aware of this performance isolation versus utilization tradeoff when we designed Snowflake, recent trends are pushing us to revisit this design choice. Specifically, maintaining a pool of pre-warmed instances was cost-efficient when infrastructure providers used to charge at an hourly granularity; however, recent move to per-second pricing [6] by all major cloud infrastructure providers has raised interesting challenges. From our (provider's) perspective, we would like to exploit this finer-grained pricing model to cut down operational costs. However doing so is not straightforward, as this trend has also led to an increase in customer-demand for finer-grained pricing. As a result, maintaining a pre-warmed pool of nodes for elasticity is no longer cost-effective: previously in the hourly billing model, as long as at least one customer VW used a particular node during a one hour duration, we could charge that customer for the entire duration. However, with per-second billing, we cannot charge unused cycles on pre-warmed nodes to any particular customer. This cost-inefficiency makes a strong case for moving to a sharing based model, where compute and ephemeral storage resources are shared across customers: in such a model we can provide elasticity by statistically multiplexing customer demands across a shared set of resources, avoiding the need to maintain a large pool of pre-warmed nodes. In the next subsection, we highlight several technical challenges that need to be resolved to realize such a shared architecture.

7.1 Resource Sharing

The variability in resource usage over time across VW, as shown in Figure 11, indicates that several of our customer workloads are bursty in nature. Hence, moving to a shared architecture would enable Snowflake to achieve better resource utilization via fine-grained statistical multiplexing. Snowflake today exposes VW sizes to customers in abstract "T-shirt" sizes (small, large, XL etc.), each representing different resource capacities. Customers are not aware of how these VWs are implemented (no. of nodes used, instance types, etc.). Ideally we would like to maintain the same abstract VW interface to customers and change the underlying implementation to use shared resources instead of isolated nodes.

The challenge, however, is to achieve isolation properties close to our current architecture. The key metric of interest from customers' point of view is query performance, that is, end-to-end query completion times. While a purely shared architecture is likely to provide good average-case performance, maintaining good performance at tail is challenging. The two key resources that need to be isolated in VWs are compute and ephemeral storage. There has been a lot of work [18, 35, 36] on compute isolation in the data center context,

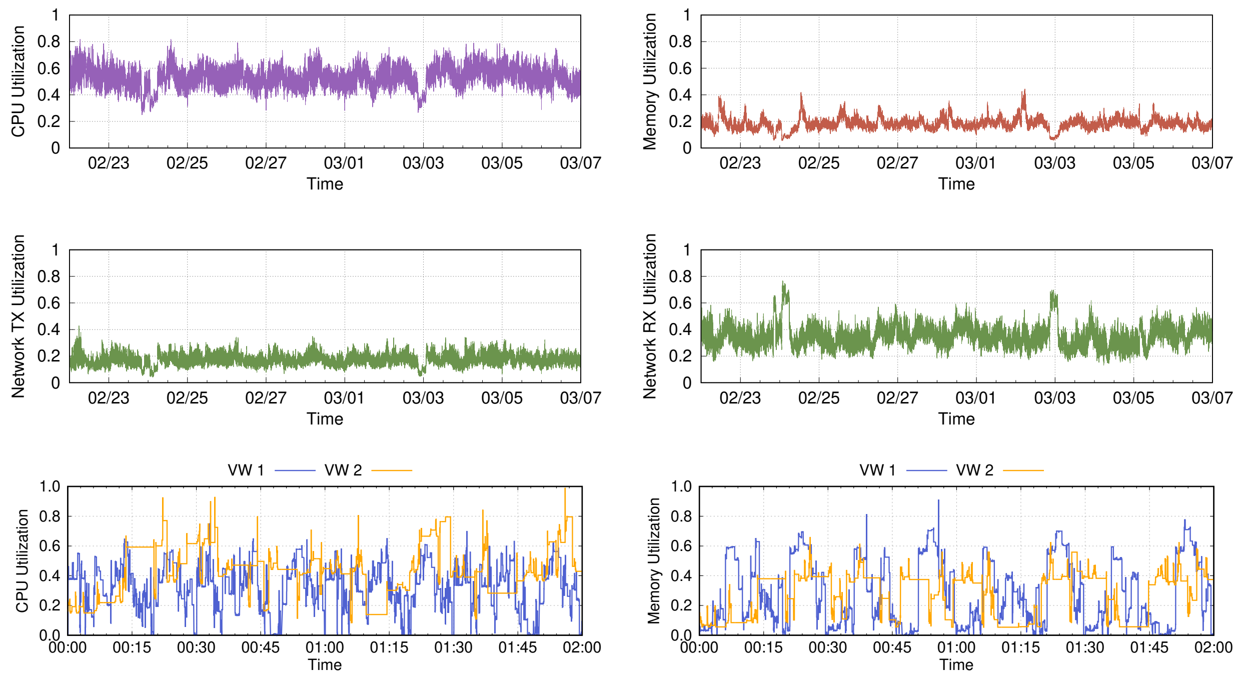


Figure 10: **System-wide CPU, Memory, and Network TX/RX utilization over time, averaged across all VWs. (top four)**. Average CPU, Memory, Network TX and Network RX utilizations are roughly 51%, 19%, 11%, 32%, respectively, indicating that there is significant room for improvement. **(bottom two)** zoomed-in CPU and Memory utilization for two highly active VW over a 2 hour duration. At several points, we see that one of the warehouses experiences high utilization while the other sees low utilization.

that Snowflake could leverage. Moreover, the centralized task scheduler and uniform execution runtime in Snowflake make the problem easier than that of isolating compute in general purpose clusters. Here, we instead focus on the problem of isolating memory and storage, which has only recently started to receive attention in the research community [25].

The goal here is to design a shared ephemeral storage system (using both memory and SSDs) that supports fine-grained elasticity without sacrificing isolation properties across tenants. With respect to sharing and isolation of ephemeral storage, we outline two key challenges. First, since our ephemeral storage system multiplexes both cached persistent data and intermediate data, both of these entities need to be jointly shared while ensuring cross-tenant isolation. While Snowflake could leverage techniques from existing literature [11, 26] for sharing cache, we need a mechanism that is additionally aware of the co-existence of intermediate data. Unfortunately, predicting the effective lifetime of cache entries is difficult. Evicting idle cache entries from tenants and providing them to other tenants while ensuring hard isolation is not possible, as we cannot predict when a tenant will next access the cache entry. Some past works [11, 33] have used techniques like idle-memory taxation to deal with this issue. We believe there is more work to be done, both in defining more reasonable isolation guarantees and designing lifetime-aware cache sharing mechanisms that can provide such guarantees.

The second challenge is that of achieving elasticity without cross-tenant interference: scaling up the shared ephemeral storage system capacity in order to meet the demands of a particular customer should not impact other tenants sharing the system. For example, if we were to naïvely use Snowflake's current ephemeral storage system, isolation properties will be trivially violated. Since all cache entries in Snowflake are consistently hashed onto the same global address space, scaling up the ephemeral storage system capacity would end up triggering the lazy consistent hashing mechanism for all tenants. This may result in multiple tenants seeing increased cache misses, resulting in degraded performance. Resolving this challenge would require the ephemeral storage system to provide private address spaces to each individual tenant, and upon scaling of resources, to reorganize data only for those tenants that have been allocated additional resources.

Memory Disaggregation. Average memory utilization in our VWs is low (Figure 10); this is particularly concerning since DRAM is expensive. Although sharing resource sharing would improve CPU and memory utilization, it is unlikely to lead to optimal utilization across both dimensions. Further, variability characteristics of CPU and memory are significantly different (Figure 11), indicating the need for independent scaling of these resources. Memory disaggregation [1, 14, 15] provides a fundamental solution to this problem. However, as discussed in §4.2, accurately provisioning re-

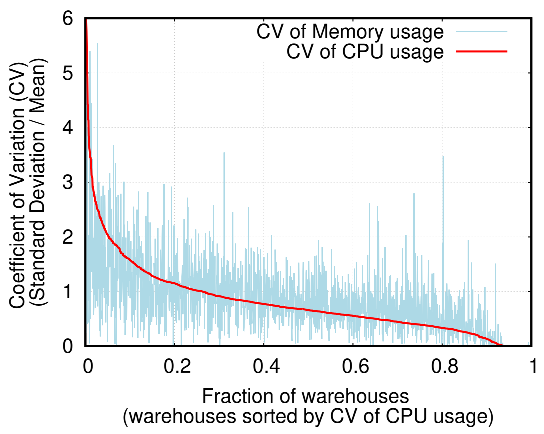


Figure 11: **Coefficient of Variation (CV) of CPU and memory usage over time, across customer VWs.** We see significant variability with respect to both resources.

sources is hard; since over-provisioning memory is expensive, we need efficient mechanisms to share disaggregated memory across multiple tenants while providing isolation guarantees.

8 Related Work

In this section we discuss related work and other systems similar to Snowflake. Our previous work [12] discusses SQL-related aspects of Snowflake and presents related literature on those aspects. This paper focuses on the disaggregation, ephemeral storage, caching, task scheduling, elasticity and multi-tenancy aspects of Snowflake; in the related work discussion below, we primarily focus on these aspects.

SQL-as-a-Service systems. There are several other systems that offer SQL functionality as a service in the cloud. These include Amazon Redshift [16], Aurora [4], Athena [3], Google BigQuery [30] and Microsoft Azure Synapse Analytics [24]. While there are papers that describe the design and operational experience of some of these systems, we are not aware of any prior work that undertakes a data-driven analysis of workload and system characteristics similar to ours.

Redshift [16] stores primary replicas of persistent data within compute VM clusters (S3 is only used for backup); thus, it may not be able to achieve the benefits that Snowflake achieves by decoupling compute from persistent storage. Aurora [4] and BigQuery [30] (based on the architecture of Dremel [23]) decouple compute and persistent storage similar to Snowflake. Aurora, however, relies on a custom-designed persistent storage service that is capable of offloading database log processing, instead of a traditional blob store. We are not aware of any published work that describes how BigQuery handles elasticity and multi-tenancy.

Decoupling compute and ephemeral storage systems. Previous work [20] makes the case for flash storage disaggregation by studying a key-value store workload from Facebook. Our observations corroborate this argument and further extend it in the context of data warehousing workloads. Pocket [22] and Locus [27] are ephemeral storage systems designed for serverless analytics applications. If we were to disaggregate compute and ephemeral storage in Snowflake, such systems would be good candidates. However, these systems do not provide fine-grained resource elasticity during the lifetime of a query. Thus, they either have to assume a priori knowledge of intermediate data sizes (for provisioning resources at the time of submitting queries), or suffer from performance degradation if such knowledge is not available in advance. As discussed in §4.1, predicting intermediate data sizes is extremely hard. It would be nice to extend these systems to provide fine-grained elasticity and cross-query isolation. Technologies for high performance access to remote flash storage [13, 17, 21] would also be integral to efficiently realize decoupling of compute and ephemeral storage system.

Multi-tenant resource sharing. ESX server [33] pioneered techniques for multi-tenant memory sharing in the virtual machine context, including ballooning and idle-memory taxation. Memshare [11] considers multi-tenant sharing of cache capacity in DRAM caches in the single machine context, sharing un-reserved capacity among applications in a way that maximizes hit rate. FairRide [26] similarly considers multi-tenant cache sharing in the distributed setting while taking into account sharing of data between tenants. Mechanisms for sharing and isolation of cache resources similar to the ones used in these works would be important in enabling Snowflake to adopt a resource shared architecture. As discussed previously, it would be interesting to extend these mechanisms to make them aware of the different characteristics and requirements of intermediate and persistent data.

9 Conclusion

We have presented operational experience running *Snowflake*, a data warehousing system with state-of-the-art SQL support. The key design and implementation aspects that we have covered in the paper relate to how Snowflake achieves compute and storage elasticity, as well as high-performance in a multi-tenancy setting. As Snowflake has grown to serve thousands of customers executing millions of queries on petabytes of data every day, we consider ourselves at least partially successful. However, using data collected from various components of our system during execution of $\sim$70 million queries over a 14 day period, our study highlights some of the shortcomings of our current design and implementation and highlights new research challenges that may be of interest to the broader systems and networking communities.

Acknowledgments

We would like to thank our shepherd, Asaf Cidon, and the anonymous NSDI reviewers for their insightful feedback. This work was supported in part by NSF 1704742 and a Google Faculty Research Award.

References

[1] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *SOCC*, 2017.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[3] Amazon. Amazon Athena. "https://aws.amazon.com/athena/".

[4] Amazon. Amazon Aurora Serverless. "https://aws.amazon.com/rds/aurora/serverless/".

[5] Amazon. Amazon simple storage service (S3). "http://aws.amazon.com/s3/".

[6] Amazon. Per-second billing for EC2 instances. "https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/".

[7] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[8] M. Azure. Azure blob storage. "https://azure.microsoft.com/en-us/services/storage/blobs/".

[9] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD*, 1997.

[11] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *ATC*, 2017.

[12] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The snowflake elastic data warehouse. In *SIGMOD*, 2016.

[13] N. Express. Nvme over fabrics overview. "https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf".

[14] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[15] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, 2017.

[16] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *SIGMOD*, 2015.

[17] J. Hwang, Q. Cai, R. Agarwal, and A. Tang. I10: A remote storage i/o stack for high-performance network and storage hardware. In *NSDI*, 2020.

[18] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *ATC*, 2018.

[19] D. Karger and M. Ruhl. New algorithms for load balancing in peer-to-peer systems. 2003.

[20] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar. Flash storage disaggregation. In *EuroSys*, 2016.

[21] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote flash $\approx$ local flash. In *ASPLOS*, 2017.

[22] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

[23] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *VLDB*, 2010.

[24] Microsoft. Azure synapse analytics. "https://azure.microsoft.com/en-us/services/synapse-analytics/".

[25] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *NSDI*, 2017.

[26] Q. Pu and H. Li. Fairride: Near-optimal, fair cache sharing. In *NSDI*, 2016.

[27] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *NSDI*, 2019.

[28] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[29] R. J. Santos and J. Bernardino. Real-time data warehouse loading methodology. In *IDEAS*, 2009.

[30] K. Sato. An inside look at google bigquery. "https://cloud.google.com/files/BigQueryTechnicalWP.pdf".

[31] A. Simitsis, P. Vassiliadis, and T. Sellis. Optimizing etl processes in data warehouses. In *ICDE*, 2005.

[32] P. Vassiliadis. A survey of extract–transform–load technology. *IJDWM*, 2009.

[33] C. A. Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 2002.

[34] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *ATC*, 2018.

[35] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multi-threading. In *ATC*, 2016.

[36] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *EuroSys*, 2013.

Millions of Tiny Databases

Marc Brooker
Amazon Web Services

Tao Chen
Amazon Web Services

Fan Ping
Amazon Web Services

Abstract

Starting in 2013, we set out to build a new database to act as the configuration store for a high-performance cloud block storage system (Amazon EBS).This database needs to be not only highly available, durable, and scalable but also strongly consistent. We quickly realized that the constraints on availability imposed by the CAP theorem, and the realities of operating distributed systems, meant that we didn't want one database. We wanted millions. Physalia is a transactional key-value store, optimized for use in large-scale cloud control planes, which takes advantage of knowledge of transaction patterns and infrastructure design to offer both high availability and strong consistency to millions of clients. Physalia uses its knowledge of datacenter topology to place data where it is most likely to be available. Instead of being highly available for all keys to all clients, Physalia focuses on being extremely available for only the keys it knows each client needs, from the perspective of that client.

This paper describes Physalia in context of Amazon EBS, and some other uses within Amazon Web Services. We believe that the same patterns, and approach to design, are widely applicable to distributed systems problems like control planes, configuration management, and service discovery.

1 Introduction

Traditional architectures for highly-available systems assume that infrastructure failures are statistically independent, and that it is extremely unlikely for a large number of servers to fail at the same time. Most modern system designs are aware of broad failure domains (data centers or availability zones), but still assume two modes of failure: a complete failure of a datacenter, or a random uncorrelated failure of a server, disk or other infrastructure. These assumptions are reasonable for most kinds of systems. Schroder and Gibson found [51] that (in traditional datacenter environments), while the probability of a second disk failure in a week was up to 9x higher when a first failure had already occurred, this correlation drops off to less than 1.5x as systems age. While a 9x higher failure rate within the following week indicates some correlation, it is still very rare for two disks to fail at the same time. This is just as well, because systems like RAID [43] and primary-backup failover perform well when failures are independent, but poorly when failures occur in bursts.

When we started building AWS in 2006, we measured the availability of systems as a simple percentage of the time that the system is available (such as 99.95%), and set Service Level Agreements (SLAs) and internal goals around this percentage. In 2008, we introduced AWS EC2 Availability Zones: named units of capacity with clear expectations and SLAs around correlated failure, corresponding to the datacenters that customers were already familiar with. Over the decade since, our thinking on failure and availability has continued to evolve, and we paid increasing attention to *blast radius* and correlation of failure. Not only do we work to make outages rare and short, we work to reduce the number of resources and customers that they affect [55], an approach we call *blast radius reduction*. This philosophy is reflected in everything from the size of our datacenters [30], to the design of our services, to operational practices.

Amazon Elastic Block Storage (EBS) is a block storage service for use with AWS EC2, allowing customers to create block devices on demand and attach them to their AWS EC2 instances. volumes are designed for an annual failure rate (AFR) of between 0.1% and 0.2%, where failure refers to a complete or partial loss of the volume. This is significantly lower than the AFR of typical disk drives [44]. EBS achieves this higher durability through replication, implementing a chain replication scheme (similar to the one described by van Renesse, et al [54]). Figure 1 shows an abstracted, simplified, architecture of EBS in context of AWS EC2. In normal operation (of this simplified model), replicated data flows through the chain from client, to primary, to replica, with no need for coordination. When failures occur, such as the failure of the primary server, this scheme requires the services of a *configuration master*, which ensures that updates to the order and membership of the replication group occur atomically, are

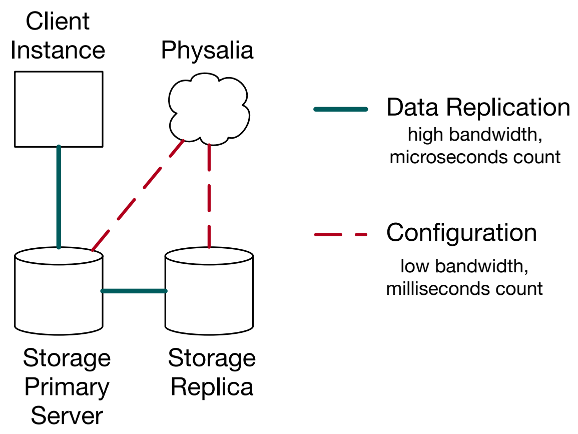


Figure 1: Simplified model of one EBS volume connected to an AWS EC2 instance. For the volume to be available for IO, either both the master and replica, or either storage server and Physalia, must be available to the instance.

well ordered, and follow the rules needed to ensure durability.

The requirements on this *configuration master* are unusual. In normal operation it handles little traffic, as replication continues to operate with no need to contact the *configuration master*. However, when large-scale failures (such as power failures or network partitions) happen, a large number of servers can go offline at once, requiring the *master* to do a burst of work. This work is latency critical, because volume IO is blocked until it is complete. It requires strong consistency, because any eventual consistency would make the replication protocol incorrect. It is also most critical at the most challenging time: during large-scale failures.

Physalia is a specialized database designed to play this role in EBS, and other similar systems at Amazon Web Services. Physalia offers both consistency and high availability, even in the presence of network partitions, as well as minimized blast radius of failures. It aims to fail gracefully and partially, and strongly avoid large-scale failures.

1.1 History

On 21 April 2011, an incorrectly executed network configuration change triggered a condition which caused 13% of the EBS volumes in a single Availability Zone (AZ) to become unavailable. At that time, replication configuration was stored in the EBS control plane, sharing a database with API traffic. From the public postmortem [46]:

> When data for a volume needs to be re-mirrored, a negotiation must take place between the AWS EC2 instance, the EBS nodes with the volume data, and the EBS control plane (which acts as an authority in this process) so that only one copy of the data is designated as the primary replica and recognized by the AWS EC2 instance as the place where all accesses should be sent. This provides strong consistency of EBS volumes. As more EBS nodes continued to fail because of the race condition described above, the volume of such negotiations with the EBS control plane increased. Because data was not being successfully re-mirrored, the number of these calls increased as the system retried and new requests came in. The load caused a brown out of the EBS control plane and again affected EBS APIs across the Region.

This failure vector was the inspiration behind Physalia's design goal of limiting the *blast radius* of failures, including overload, software bugs, and infrastructure failures.

1.2 Consistency, Availability and Partition Tolerance

As proven by Gilbert and Lynch [22], it is not possible for a distributed system to offer both strong consistency (in the sense of linearizability [31]), and be available to all clients in the presence of network partitions. Unfortunately, all real-world distributed systems must operate in the presence of network partitions [6], so systems must choose between strong consistency and availability. Strong consistency is non-negotiable in Physalia, because it's required to ensure the correctness of the EBS replication protocol. However, because chain replication requires a configuration change during network partitions, it is especially important for Physalia to be available during partitions.

Physalia then has the goal of optimizing for availability during network partitions, while remaining strongly consistent. Our core observation is that we do not require all keys to be available to all clients. In fact, each key needs to be available at only three points in the network: the AWS EC2 instance that is the client of the volume, the primary copy, and the replica copy. Through careful placement, based on our system's knowledge of network and power topology, we can significantly increase the probability that Physalia is available to the clients that matter for the keys that matter to those clients.

This is Physalia's key contribution, and our motivation for building a new system from the ground up: infrastructure aware placement and careful system design can significantly reduce the effect of network partitions, infrastructure failures, and even software bugs. In the same spirit as Paxos Made Live [12], this paper describes the details, choices and tradeoffs that are required to put a consensus system into production. Our concerns, notably blast radius reduction and infrastructure awareness, are significantly different from that paper.

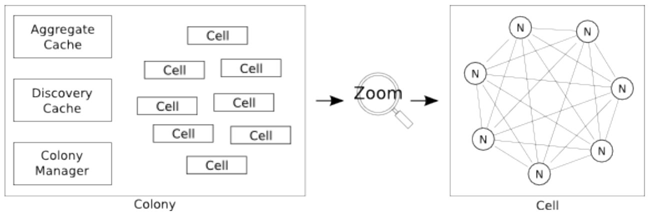


Figure 2: Overview of the relationship between the colony, cell and node.

2 The Design of Physalia

Physalia's goals of blast radius reduction and partition tolerance required careful attention in the design of the data model, replication mechanism, cluster management and even operational and deployment procedures. In addition to these top-level design goals, we wanted Physalia to be easy and cheap to operate, contributing negligibly to the cost of our dataplane. We wanted its data model to be flexible enough to meet future uses in similar problem spaces, and to be easy to use correctly. This goal was inspired by the concept of *misuse resistance* from cryptography (GCM-SIV [27], for example), which aims to make primitives that are safer under misuse. Finally, we wanted Physalia to be highly scalable, able to support an entire EBS availability zone in a single installation.

2.1 Nodes, Cells and the Colony

The Portuguese man o' war (*Physalia physalis*) is not one animal, but a siphonophore: a colonial organism made up of many specialized animals called zooids. These zooids are highly adapted to living in the colony, and cannot live outside it. Nevertheless, each zooid is a stand-alone organism, including everything that is required for life. Physalia's high-level organization is similar: each Physalia installation is a *colony*, made up of many *cells*. The *cells* live in the same environment: a mesh of *nodes*, with each node running on a single server. Each *cell* manages the data of a single partition key, and is implemented using a distributed state machine, distributed across seven nodes. Cells do not coordinate with other cells, but each node can participate in many cells. The colony, in turn, can consist of any number of cells (provided there are sufficient nodes to distribute those cells over). Figure 2 captures the relationship between colony, cell and node. Figure 3 shows the cell: a mesh of nodes holding a single Paxos-based distributed state machine, with one of the nodes playing the role of *distinguished proposer*.

The division of a colony into a large number of cells is our main tool for reducing radius in Physalia. Each node is only used by a small subset of cells, and each cell is only used by a small subset of clients.

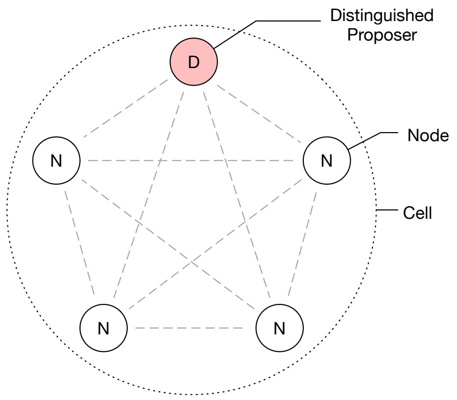


Figure 3: A cell is a group of nodes, one of which assumes the role of distinguished proposer.

Each Physalia colony includes a number of control plane components. The control plane plays a critical role in maintaining system properties. When a new cell is created, the control plane uses its knowledge of the power and network topology of the datacenter (discovered from AWS's datacenter automation systems) to choose a set of nodes for the cell. The choice of nodes balances two competing priorities. Nodes should be placed close to the clients (where *close* is measured in logical distance through the network and power topology) to ensure that failures *far away* from their clients do not cause the cell to fail. They must also be placed with sufficient diversity to ensure that small-scale failures do not cause the cell to fail. Section 3 explores the details of placement's role in availability.

The cell creation and repair workflows respond to requests to create new cells (by placing them on under-full nodes), handling cells that contain failed nodes (by replacing these nodes), and moving cells closer to their clients as clients move (by incrementally replacing nodes with closer ones).

We could have avoided implementing a seperate control-plane and repair workflow for Physalia, by following the example of elastic replication [2] or Scatter [23]. We evaluated these approaches, but decided that the additional complexity, and additional communication and dependencies between shards, were at odds with our focus on blast radius. We chose to keep our cells completely independent, and implement the control plane as a seperate system.

2.2 Physalia's Flavor of Paxos

The design of each cell is a straightforward consensus-based distributed state machine. Cells use Paxos [35] to create an ordered log of updates, with batching and pipelining [48] to improve throughput. Batch sizes and pipeline depths are kept small, to keep per-item work well bounded and ensure short time-to-recovery in the event of node or network failure. Physalia uses a custom implementation of Paxos written

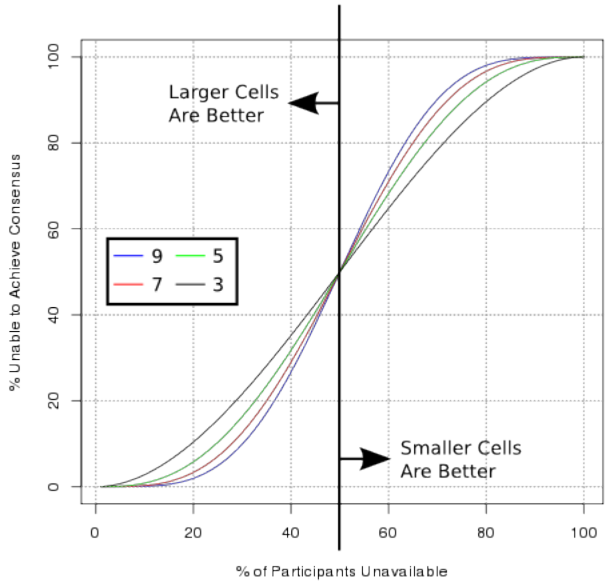


Figure 4: The size of cells is a trade-off between tolerance to large correlated failures and tolerance to random failures.

in Java, which keeps all required state both in memory and persisted to disk. In typical cloud systems, durability is made easier by the fact that systems can be spread across multiple datacenters, and correlated outages across datacenters are rare. Physalia's locality requirement meant that we could not use this approach, so extra care in implementation and testing were required to ensure that Paxos is implemented safely, even across dirty reboots.

In the EBS installation of Physalia, the cell performs Paxos over seven nodes. Seven was chosen to balance several concerns:

- *Durability* improves exponentially with larger cell size [29]. Seven replicas means that each piece of data is durable to at least four disks, offering durability around 5000x higher than the 2-replication used for the volume data.

- Cell size has little impact on mean *latency*, but larger cells tend to have lower high percentiles because they better reject the effects of slow nodes, such as those experiencing GC pauses [17].

- The effect of cell size on *availability* depends on the type of failures expected. As illustrated in Figure 4, smaller cells offer lower availability in the face of small numbers of uncorrelated node failures, but better availability when the proportion of node failure exceeds 50%. While such high failure rates are rare, they do happen in practice, and a key design concern for Physalia.

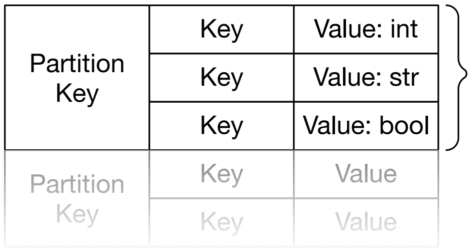


Figure 5: The Physalia schema.

- Larger cells consume more resources, both because Paxos requires $O(\text{cellsize})$ communication, but also because a larger cell needs to keep more copies of the data. The relatively small transaction rate, and very small data, stored by the EBS use of Physalia made this a minor concern.

The control plane tries to ensure that each node contains a different mix of cells, which reduces the probability of correlated failure due to load or poison pill transitions. In other words, if a poisonous transition crashes the node software on each node in the cell, only that cell should be lost. In the EBS deployment of Physalia, we deploy it to large numbers of nodes well-distributed across the datacenter. This gives the Physalia control plane more placement options, allowing it to optimize for widely-spread placement.

In our Paxos implementation, proposals are accepted optimistically. All transactions given to the proposer are proposed, and at the time they are to be applied (i.e. all transactions with lower log positions have already been applied), they are committed or ignored depending on whether the write conditions pass. The advantage of this optimistic approach is that the system always makes progress if clients follow the typical optimistic concurrency control (OCC) pattern. The disadvantage is that the system may do significant additional work during contention, passing many proposals that are never committed.

2.3 Data Model and API

The core of the Physalia data model is a partition key. Each EBS volume is assigned a unique partition key at creation time, and all operations for that volume occur within that partition key. Within each partition key, Physalia offers a transactional store with a typed key-value schema, supporting strict serializable reads, writes and conditional writes over any combination of keys. It also supports simple in-place operations like atomic increments of integer variables. Figure 5 shows the schema: one layer of partition keys, any number (within operational limitations) of string keys within a partition, and one value per key. The API can address only one partition key at a time, and offers strict serializable batch and conditional operations within the partition.

The goal of the Physalia API design was to balance two competing concerns. The API needed to be expressive enough for clients to take advantage of the (per-cell) transactional nature of the underlying store, including expressing conditional updates, and atomic batch reads and writes. Increasing API expressiveness, on the other hand, increases the probability that the system will be able to accept a transition that cannot be applied (a poison pill). The Physalia API is inspired by the Amazon DynamoDB API, which supports atomic batched and single reads and writes, conditional updates, paged scans, and some simple in-place operations like atomic increments. We extended the API by adding a compound read-and-conditional-write operation.

Phsyalia's data fields are strong but dynamically typed. Supported field types include byte arrays (typically used to store UTF-8 string data), arbitrary precision integers, and booleans. Strings are not supported directly, but may be offered as a convenience in the client. Floating-point data types and limited-precision integers are not supported due to difficulties in ensuring that nodes will produce identical results when using different software versions and hardware (see [24] and chapter 11 of [1]). As in any distributed state machine, it's important that each node in a cell gets identical results when applying a transition. We chose not to offer a richer API (like SQL) for a similar reason: our experience is that it takes considerable effort to ensure that complex updates are applied the same way by all nodes, across all software versions.

Physalia provides two consistency modes to clients. In the consistent mode, read and write transactions are both linearizable and serializable, due to being serialized through the state machine log. Most Physalia clients use this consistent mode. The eventually consistent mode supports only reads (all writes are consistent), and offers a consistent prefix [7] to all readers and monotonic reads [53] within a single client session. Eventually consistent reads are provided to be used for monitoring and reporting (where the extra cost of linearizing reads worth it), and the discovery cache (which is eventually consistent anyway).

The API also offers first-class leases [25] (lightweight time-bounded locks). The lease implementation is designed to tolerate arbitrary clock skew and short pauses, but will give incorrect results if long-term clock rates are too different. In our implementation, this means that the fastest node clock is advancing at more than three times the rate of the slowest clock. Despite lease safety being highly likely, leases are only used where they are not critical for data safety or integrity.

In the Physalia API, all keys used to read and write data, as well as conditions for conditional writes, are provided in the input transaction. This allows the proposer to efficiently detect which changes can be safely batched in a single transaction without changing their semantics. When a batch transaction is rejected, for example due to a conditional put failure, the proposer can remove the offending change from the batch and re-submit, or submit those changes without batching.

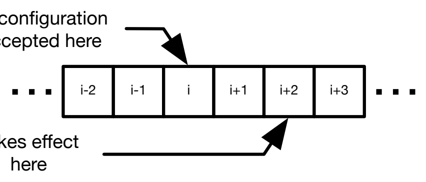


Figure 6: Changes in membership are placed into the log, but only take effect some time later (pictured here is $\alpha = 2$)

2.4 Reconfiguration, Teaching and Learning

As with our core consensus implementation, Physalia does not innovate on reconfiguration. The approach taken of storing per-cell configuration in the distributed state machine and passing a transition with the existing jury to update it follows the pattern established by Lampson [37]. A significant factor in the complexity of reconfiguration is the interaction with pipelining: configuration changes accepted at log position i must not take effect logically until position $i+\alpha$, where α is the maximum allowed pipeline length (illustrated in Figure 6). Physalia keeps α small (typically 3), and so simply waits for natural traffic to cause reconfiguration to take effect (rather than stuffing no-ops into the log). This is a very sharp edge in Paxos, which doesn't exist in either Raft [42] or Viewstamped Replication [41].

Physalia is unusual in that reconfiguration happens frequently. The colony-level control plane actively moves Physalia cells to be close to their clients. It does this by replacing far-away nodes with close nodes using reconfiguration. The small data sizes in Physalia make cell reconfiguration an insignificant portion of overall datacenter traffic. Figure 7 illustrates this process of movement by iterative reconfiguration. The system prefers safety over speed, moving a single node at a time (and waiting for that node to catch up) to minimize the impact on durability. The small size of the data in each cell allows reconfiguration to complete quickly, typically allowing movement to complete within a minute.

When nodes join or re-join a cell they are brought up to speed by *teaching*, a process we implement outside the core consensus protocol. We support three modes of teaching. In the bulk mode, most suitable for new nodes, the teacher (any existing node in the cell) transfers a bulk snapshot of its state machine to the learner. In the log-based mode, most suitable for nodes re-joining after a partition or pause, the teacher ships a segment of its log to the learner. We have found that this mode is triggered rather frequently in production, due to nodes temporarily falling behind during Java garbage collection pauses. Log-based learning is chosen when the size of the missing log segment is significantly smaller than the size of the entire dataset.

Finally, packet loss and node failures may leave persistent holes in a node's view of the log. If nodes are not able to find

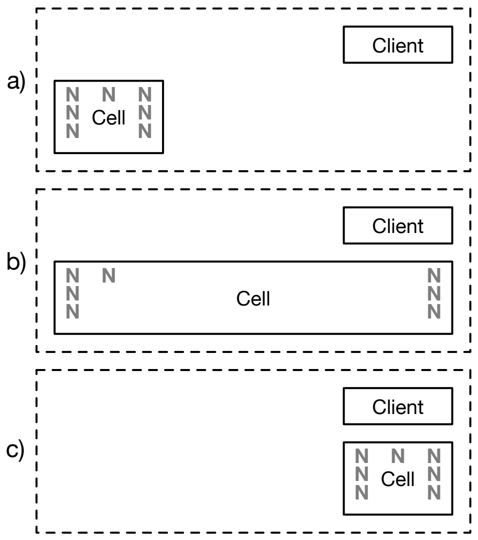


Figure 7: When Physalia detects that a cell's client has moved (a), it replaces nodes in the cell with ones closer to the client (b), until the cell is entirely nearby the client (c).

another to teach them the decided value in that log position (or no value has been decided), they use a whack-a-mole learning mode. In whack-a-mole mode, a learner actively tries to propose a no-op transition into the vacant log position. This can have two outcomes: either the acceptors report no other proposals for that log position and the no-op transition is accepted, or another proposal is found and the learner proposes that value. This process is always safe in Paxos, but can affect liveness, so learners apply substantial jitter to whack-a-mole learning.

2.5 The Discovery Cache

Clients find cells using a distributed *discovery cache*. The *discovery cache* is a distributed eventually-consistent cache which allow clients to discover which nodes contain a given cell (and hence a given partition key). Each cell periodically pushes updates to the cache identifying which partition key they hold and their node members. Incorrect information in the cache affects the liveness, but never the correctness, of the system. We use three approaches to reduce the impact of the *discovery cache* on availability: client-side caching, forwarding pointers, and replication. First, it is always safe for a client to cache past discovery cache results, allowing them to refresh lazily and continue to use old values for an unbounded period on failure. Second, Physalia nodes keep long-term (but not indefinite) forwarding pointers when cells move from node to node. Forwarding pointers include pointers to all the nodes in a cell, making it highly likely that a client will succeed in pointer chasing to the current owner provided that it can get to at least one of the past owners. Finally, because the discovery cache is small, we can economically keep many copies of it, increasing the probability that at least one will be available.

2.6 System Model and Byzantine Faults

In designing Physalia, we assumed a system model where messages can be arbitrarily lost, replayed, re-ordered, and modified after transmission. Message authentication is implemented using a cryptographic HMAC on each message, guarding against corruption occurring in lower layers. Messages which fail authentication are simply discarded. Key distribution, used both for authentication and prevention of unintentional Sybil-style attacks [20] is handled by our environment (and therefore out of the scope of Physalia), optimising for frequent and low-risk key rotation.

This model extends the "benign faults" assumptions of Paxos [11] slightly, but stops short of Byzantine fault tolerance[1]. While Byztantine consensus protocols are well understood, they add significant complexity to both software and system interactions, as well as testing surface area. Our approach was to keep the software and protocols simpler, and mitigate issues such as network and storage corruption with cryptographic integrity and authentication checks at these layers.

3 Availability in Consensus Systems

State-machine replication using consensus is popular approach for building systems that tolerate faults in single machines, and uncorrelated failures of a small number of machines. In theory, systems built using this pattern can achieve extremely high availability. In practice, however, achieving high availability is challenging. Studies across three decades (including Gray in 1990 [26], Schroeder and Gibson in 2005 [50] and Yuan et al in 2014 [57]) have found that software, operations, and scale drive downtime in systems designed to tolerate hardware faults. Few studies consider a factor that is especially important to cloud customers: large-scale correlated failures which affect many cloud resources at the same time.

3.1 Physalia vs the Monolith

It is well known that it is not possible to offer both all-clients availability and consistency in distributed databases due to the presence of network partitions. It is, however, possible to offer both consistency and availability to clients on the majority side of a network partition. While long-lived network partitions are rare in modern datacenter networks, they do occur, both due to the network itself and other factors (see Bailis and Kingsbury [6] and Alquraan et al [5] for surveys of

[1]This approach is typical of production consensus-based systems, including popular open-source projects like Zookeeper and etcd

causes of network partitions). Short-lived partitions are more frequent. To be as available as possible to its clients, Physalia needs to be on the same side of any network partition as them. For latency and throughput reasons, EBS tries to keep the storage replicas of a volume close to the AWS EC2 instances the volumes are attached to, both in physical distance and network distance. This means that client, data master and data replica are nearby each other on the network, and Physalia needs to be nearby too. Reducing the number of network devices between the Physalia database and its clients reduces the possibility of a network partition forming between them for the simple reason that fewer devices means that there's less to go wrong.

Physalia also optimizes for blast radius. We are not only concerned with the availability of the whole system, but want to avoid failures of the whole system entirely. When failures happen, due to any cause, they should affect as small a subset of clients as possible. Limiting the number of cells depending on a single node, and clients on a single cell, significantly reduce the effect that one failure can have on the overall system.

This raises the obvious question: does Physalia do better than a monolithic system with the same level of redundancy? A monolithic system has the advantage of less complexity. No need for the discovery cache, most of the control plane, cell creation, placement, etc. Our experience has shown that simplicity improves availability, so this simplification would be a boon. On the other hand, the monolithic approach loses out on partition tolerance. It needs to make a trade-off between being localized to a small part of the network (and so risking being partitioned away from clients), or being spread over the network (and so risking suffering an internal partition making some part of it unavailable). The monolith also increases blast radius: a single bad software deployment could cause a complete failure (this is similar to the node count trade-off of Figure 4, with one node).

3.2 Placement For Availability

The EBS control plane (of which the Physalia control plane is a part) continuously optimizes the availability of the EBS volume $P(A_{\mathrm{v}})$ to the client AWS EC2 instance, and the EBS storage servers that store the volume. This is most interesting to do when the client instance is available. If the volume is unavailable at the same time as the client instance, we know that the instance will not be trying to access it. In other words, in terms of the availability of the volume (A_{v}), and the instance (A_{i}), the control plane optimizes the conditional probability $P(A_{\mathrm{v}}|A_{\mathrm{i}})$. The ideal solution to this problem is to entirely co-locate the volume and instance, but EBS offers the ability to detach a volume from a failed instance, and re-attach it to another instance. To make this useful, volumes must continue to be durable even if the instance suffers a failure. Placement must therefore balance the concerns of having the volume close enough for correlated availability, but far enough away for sufficiently independent durability to meet EBS's durability promise.

As an example, consider an idealized datacenter with three levels of network (servers, racks and rows) and three power domains (A, B and C). The client instance is on one rack, the primary copy on another, and replica copy on a third, all within the same row. Physalia's placement will then ensure that all nodes for the cell are within the row (there's no point being available if the row is down), but spread across at least three racks to ensure that the loss of one rack doesn't impact availability. It will also ensure that the nodes are in three different power domains, with no majority in any single domain.

This simple scheme faces two challenges. One is that real-world datacenter topology is significantly more complex, especially where datacenters contain multiple generations of design and layout. Another is that EBS volumes move by replication, and their clients move by customers detaching their volumes from one instance and attaching them to another. The Physalia control plane continuously responds to these changes in state, moving nodes to ensure that placement constraints continue to be met.

3.3 Non-Infrastructure Availability Concerns

Another significant challenge with building high-availability distributed state machines is correlated work. In a typical distributed state machine design, each node is processing the same updates and the same messages in the same order. This leads the software on the machines to be in the same state. In our experience, this is a common cause of outages in real-world systems: redundancy does not add availability if failures are highly correlated. Having all copies of the software in the same state tends to trigger the same bugs in each copy at the same time, causing multiple nodes to fail, either partially or completely, at the same time. Another issue is that the correlated loads cause memory, hard drives, and other storage on each host to fill up at the same rate. Again, this causes correlated outages when each host has the same amount of storage. Poison pill transactions may also cause outages; these are transactions that are accepted by the cell but cannot be applied to the state machine once consensus is reached.

Software deployments and configuration changes also contribute to downtime. Good software development practices, including code review and automated and manual testing, can reduce the risk of software changes but not entirely eliminate it. Incremental deployment, where code is rolled out slowly across the fleet and rolled back at the first sign of trouble, is a required operational practice for highly available systems. The fault-tolerant nature of distributed state machines makes this approach less effective: because the system is designed to tolerate failure in less than half of hosts, failure may not be evident until new code is deployed to half of all hosts. Practices like positive validation, where the deployment system checks that new nodes are taking traffic, reduce but do not eliminate this risk.

Poison pills are a particularly interesting case of software failure. A poison pill is a transaction which passes validation and is accepted into the log, but cannot be applied without causing an error. Pipelining requires that transactions are validated before the state they will execute on is fully known, meaning that even simple operations like numerical division could be impossible to apply. In our experience, poison pills are typically caused by under-specification in the transaction logic (*"what does dividing by zero do?"*, *"what does it mean to decrement an unsigned zero?"*), and are fixed by fully specifying these behaviors (a change which comes with it's own backward-compatibility challenges).

All of these factors limit the availability of any single distributed state machine, as observed by its clients. To achieve maximum availability, we need many such systems spread throughout the datacenter. This was the guiding principle of Physalia: instead of one database, build millions.

3.4 Operational Practices

Our experience of running large distributed systems is that operations, including code and configuration deployments, routine system operations such as security patching, and scaling for increased load, are dominant contributors to system downtime, despite ongoing investments in reducing operational defect rates. This conclusion isn't particular to the environment at AWS. For example, Jim Gray found in 1990 that the majority of failures of Tandem computers were driven by software and operations [26]. Operational practices at AWS already separate operational tasks by region and availability zone, ensuring that operations are not performed across many of these units at the same time.

Physalia goes a step further than this practice, by introducing the notion of *colors*. Each cell is assigned a color, and each cell is constructed only of nodes of the same color. The control plane ensures that colors are evenly spread around the datacenter, and color choice minimally constrains how close a cell can be to its clients. Physalia's very large node and cell counts make this possible. When software deployments and other operations are performed, they proceed color-by-color. Monitoring and metrics are set up to look for anomalies in single colors. Colors also provide a layer of isolation against load-related and poison pill failures. Nodes of different colors don't communicate with each other, making it significantly less likely that a poison pill or overload could spread across colors.

3.5 Load in Sometimes-Coordinating Systems

Load is another leading cause of correlated failures. Fundamentally, a consensus-based system needs to include more than half of all nodes in each consensus decision, which means that overload can take out more than half of all nodes. Colors play a role in reducing the blast radius from load spikes from a few clients, but the load on Physalia is inherently spiky.

During normal operation, load consists of a low rate of calls caused by the background rate of EBS storage server failures, and creation of new cells for new volumes. During large-scale failures, however, load can increase considerably. This is an inherent risk of *sometimes-coordinating* systems like EBS: recovery load is not constant, and highest during bad network or power conditions. See Section 5.2.1 for a brief exploration of the magnitude of these spikes.

Per-cell Physalia throughput, as is typical of Paxos-style systems, scales well up to a point, with significant wins coming from increased batch efficiency. Beyond this point, however, contention and the costs of co-ordination cause goodput to drop with increased load (as predicted by Gunther's model [28]). To avoid getting into this reduced-goodput mode, cells reject load once their pipelines are full. While this isn't a perfect predictor of load, it works well because it decreases attempted throughput with increased latency (and is therefore stable in the control theory sense), and gets close to peak system throughput. Clients are expected to exponentially back off, apply jitter, and eventually retry their rejected transactions. As the number of clients in the Physalia system is bounded, this places an absolute upper limit on load, at the cost of latency during overload.

4 Testing

The challenge of testing a system like Physalia is as large as the challenge of designing and building it. Testing needs to cover not only the happy case, but also a wide variety of error cases. Our experience mirrors the findings of Yuan, et al [57] that error handling is where many bugs hide out, and Alquraan et al [5] that network partitions are rare events that easily hide bugs. As Kingsbury's Jepsen [33] testing work has found, many consensus implementations also have bugs in the happy path. Good testing needs to look everywhere. To get the coverage required, we needed to make the bar to building a new test case extremely low.

4.1 The SimWorld

To solve this problem, we picked an approach that is in wide use at Amazon Web Services, which we would like to see broadly adopted: build a test harness which abstracts networking, performance, and other systems concepts (we call it a *simworld*). The goal of this approach is to allow developers to write distributed systems tests, including tests that simulate packet loss, server failures, corruption, and other failure cases, as *unit tests* in the same language as the system itself. In this case, these *unit tests* run inside the developer's IDE (or with *junit* at build time), with no need for test clusters or other infrastructure. A typical test which tests correctness under packet loss can be implemented in less than 10 lines of Java code, and executes in less than 100ms. The Physalia team have written hundreds of such tests, far exceeding the coverage that would be practical in any cluster-based or container-based approach.

The key to building a *simworld* is to build code against abstract physical layers (such as networks, clocks, and disks). In Java we simply wrap these thin layers in interfaces. In production, the code runs against implementations that use real TCP/IP, DNS and other infrastructure. In the *simworld*, the implementations are based on in-memory implementations that can be trivially created and torn down. In turn, these in-memory implementations include rich fault-injection APIs, which allow test implementors to specify simple statements like:

```
net.partitionOff(PARTITION_NAME, p5.
    getLocalAddress());
...
net.healPartition(PARTITION_NAME);
```

Our implementation allows control down to the packet level, allowing testers to delay, duplicate or drop packets based on matching criteria. Similar capabilities are available to test disk IO. Perhaps the most important testing capability in a distributed database is time, where the framework allows each actor to have it's own view of time arbitrarily controlled by the test. Simworld tests can even add Byzantine conditions like data corruption, and operational properties like high latency. We highly recommend this testing approach, and have continued to use it for new systems we build.

4.2 Additional Testing Approaches

In addition to unit testing, we adopted a number of other testing approaches. One of those approaches was a suite of automatically-generated tests which run the Paxos implementation through every combination of packet loss and reordering that a node can experience. This testing approach was inspired by the TLC model checker [56], and helped us build confidence that our implementation matched the formal specification.

We also used the open source Jepsen tool [33] to test the system, and make sure that the API responses are linearizable under network failure cases. This testing, which happens at the infrastructure level, was a good complement to our lower-level tests as it could exercise some under-load cases that are hard to run in the *simworld*.

Finally, we performed a number of *game days* against deployments of Physalia. A game day is a failure simulation that happens in a real production or production-like deployment of a system, an approach that has been popular at Amazon for 20 years. Game days test not only the correctness of code, but also the adequacy of monitoring and logging, effectiveness of operational approaches, and the team's understanding of how to debug and fix the system. Our game day approach is similar to the chaos engineering approach pioneered by Netflix [32], but typically focuses on larger-scale failures rather than component failures.

4.3 The Role of Formal Methods

TLA+ [36] is a specification language that's well suited to building formal models of concurrent and distributed systems. We use TLA+ extensively at Amazon [39], and it proved exceptionally useful in the development of Physalia. Our team used TLA+ in three ways: writing specifications of our protocols to check that we understand them deeply, model checking specifications against correctness and liveness properties using the TLC model checker, and writing extensively commented TLA+ code to serve as the documentation of our distributed protocols. While all three of these uses added value, TLA+'s role as a sort of automatically tested (via TLC), and extremely precise, format for protocol documentation was perhaps the most useful. Our code reviews, *simworld* tests, and design meetings frequently referred back to the TLA+ models of our protocols to resolve ambiguities in Java code or written communication. We highly recommend TLA+ (and its Pluscal dialect) for this use.

One example of a property we checked using TLA+ is the safety of having stale information in the discovery cache. For correctness, it is required that a client acting on stale information couldn't cause a *split brain* by allowing a group of old nodes to form a quorum. We started with the informal argument that the reconfiguration protocol makes $f > \frac{N}{2}$ of the pre-reconfiguration nodes aware of a configuration change, and therefore aware if they have been deposed from the jury. In other words, at most $\lfloor \frac{N}{2} \rfloor$ nodes may have been deposed from the jury without being aware of the change, and because they do not form a quorum they cannot pass *split brain* proposals. This argument becomes successively more complex as multiple reconfigurations are passed, especially during a single window of α. Multiple reconfigurations also introduce an ABA problem when cells move off, and then back onto, a node. TLA+ and TLC allowed us to build confidence in the safety of our protocols in this complex case and cases like it.

5 Evaluation

Evaluating the performance of a system like Physalia is challenging. Performance, including throughput and latency, are important, but the most important performance metrics are how the system performs during extremely rare large-scale outages. We evaluate the performance of Physalia in production, and evaluate the design through simulations. We also use simulations to explore some particularly challenging whole-system aspects of Physalia.

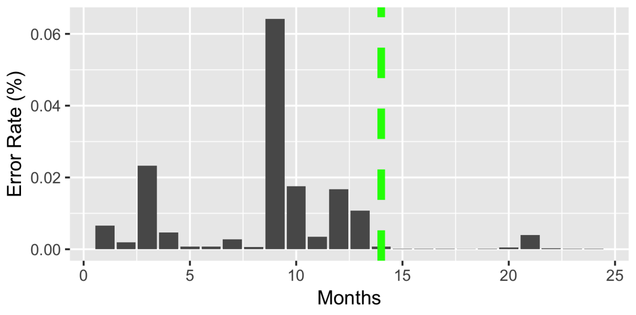


Figure 8: Mean availability of the configuration store from the perspective of the EBS primary, bucketed by month, for a production colony. The vertical line shows the deployment of Physalia in this datacenter, replacing a legacy system.

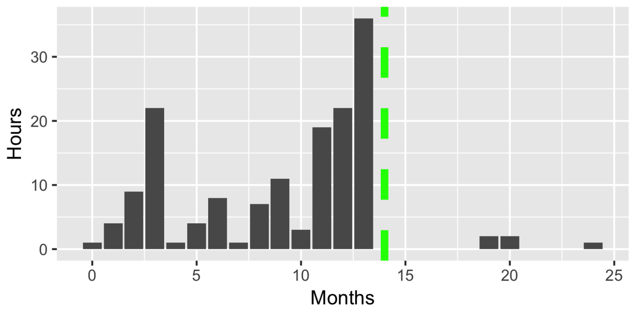


Figure 9: Number of hours per month where EBS masters experienced an error rate $> 0.05\%$ in a production colony. The vertical line shows the deployment of Physalia.

5.1 Production Experience

Physalia is deployed in production in AWS, running in over 60 availability zones. Figure 8 shows the effect that it's deployment has had on one measure of volume availability: how often the primary copy of the volume is able to contact the configuration store on the first try. The deployment of Physalia shows a clear ($p = 7.7x10^{-5}$) improvement in availability. Availability failures in the previous system were caused both by infrastructure failures and by transient overload (see Section 3.5).

Figure 9 shows the same data in a different way, looking at compliance against an internal error rate goal, significantly stricter than the external SLA for EBS. In this case, the internal goal is 0.05%, and we count the number of hours where this goal is exceeded.

In production deployments within AWS, Physalia deployments at availability-zone scale routinely serve thousands of requests per second. Latency varies between read and write optimizations. Linearizable reads can sometimes be handled by the distinguished proposer. Writes, on the other hand, need to complete a Paxos round before they are committed, and therefore require substantially more communication. Figure 10 presents a multi-day view of read and write latency percentiles, calculated on a one-minute bucket. In this typical installation, reads take less than 10ms at the 99th percentile, and writes typically take less than 50ms.

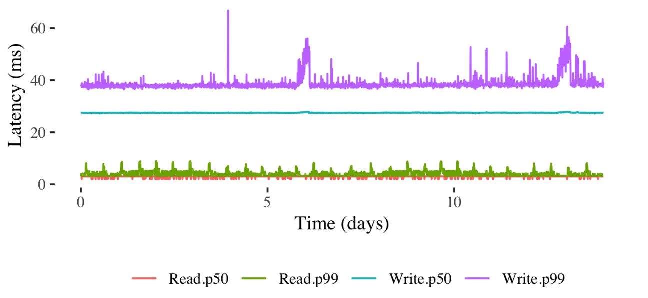


Figure 10: Physalia read and write latencies for one large-scale cluster. p50 is the 50th percentile, and p99 is the 99th.

In a distributed state machine, not only must operations be applied deterministically across all replicas, but they must be applied the same way by all production versions. Our operational and testing practices handle this edge case by testing between adjacent versions. Early in our production rollout, a bug in our deployment tools lead to a *rollback* to an old version of the code base on a small number of nodes. These nodes applied transactions differently, simply not applying a conditional they didn't understand, leading to state to diverge on the cells where they were members. While we fixed this issue quickly with little customer impact, we took three important lessons away from it. First, Postel's famous robustness principle (*be conservative in what you do, be liberal in what you accept from others*) [45] does not apply to distributed state machines: they should not accept transactions they only partially understand and allow the consensus protocol to treat them as temporarily failed. Second, our testing processes needed to cover more than adjacent versions, and include strong mechanisms for testing rollback cases (both expected and unexpected). The third lesson is perhaps the most important: control planes should exploit their central position in a systems architecture to offer additional safety. When the rollback issue occurred, affected cells were corrupted in a way that caused the control plane to see them as empty, and available for deletion. The control plane dutifully took action, deleting the cells. Based on this experience, we modified the control plane to add rate limiting logic (don't move faster than the expected rate of change), and a *big red button* (allowing operators to safely and temporarily stop the control plane from taking action). Control planes provide much of the power of the cloud, but their privileged position also means that they have to act safely, responsibly, and carefully to avoid introducing additional failures.

5.2 Design Validation via Simulation

The statistical behavior of a system as complex as Physalia can be difficult, if not intractable, to analyze in closed form. From early in the feasibility stages to production deployment, we used simulation to understand the dynamic behavior of the system, explore alternative system designs, and calculate baselines for our testing. In this section, we present some simulation results, and conclude by comparing the performance of the system to those results.

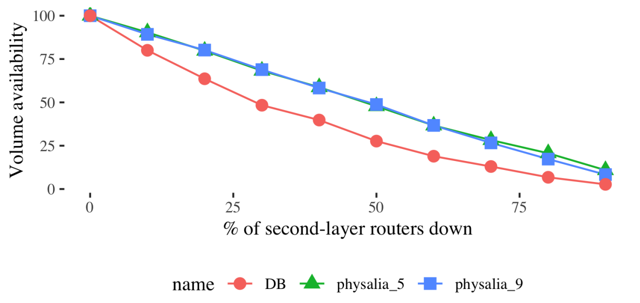
The availability offered by a Physalia deployment is highly sensitive to the failure modes of the underlying infrastructure, and the statistical properties of each of those failure modes. These results use a simplified (and outdated) model of a datacenter network: servers are organized into racks, each with a top-of-rack switch (*tor*), which in turn connects to one or more aggregation routers (*aggs*), which connect to one or more core routers (*cores*). Typical real-world networks contain some redundancy. For example, a *tor* is likely to connect to more than one *agg*. In these results we've left out redundancy for simplicity's sake, but the results are qualitatively similar (although the failure statistics are very different), once redundancy is considered.

One significant area that we explored with simulation is placement. Globally optimizing the placement of Physalia volumes is not feasible for two reasons, one is that it's a non-convex optimization problem across huge numbers of variables, the other is that it needs to be done online because volumes and cells come and go at a high rate in our production environment. Figure 11 shows the results of using one very rough placement heuristic: a sort of *bubble sort* which swaps nodes between two cells at random if doing so would improve locality. In this simulation, we considered 20 candidates per cell. Even with this simplistic and cheap approach to placement, Physalia is able to offer significantly (up to 4x) reduced probability of losing availability.

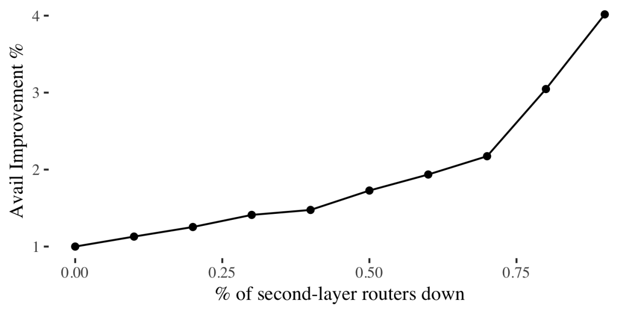
5.2.1 Simulations of System Load

As discussed in Section 3.5, load on Physalia can vary dramatically with different network conditions. Simulation of failures in different network topologies allows us to quantify the maximum expected load. Figure 12 shows the results of simulating *agg* failures (in the same model used above) on offered load to Physalia. A volume needs to call Physalia if the client AWS EC2 instance can get to either the master or replica EBS server, but the master and replica can't get to each other.

At small failure rates, expected load increases linearly with the count of failed devices, up to maximum of 29%. Beyond this, load drops off, as volumes become likely to be completely disconnected from the client. Multiplying this graph (or, ideally one simulated on actual datacenter topology) with the expected values of device failures yields a graph of the expectation of the magnitude of maximum load on Physalia (or, indeed, any configuration master in an EBS-like replicated system). These results closely match what we have observed of the real-world behavior of the EBS deployment of Physalia.



(a)



(b)

Figure 11: Simulated availability of volumes using Physalia, versus a baseline of a single-point database, under network partitions caused by device failures at the *agg* layer. (a) shows raw results for cell sizes 5 and 9, and (b) shows the ratio between Physalia and baseline availability.

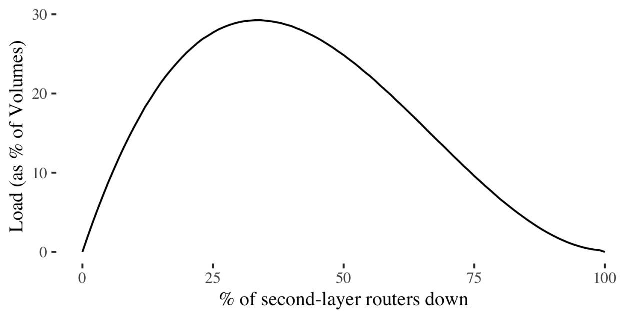


Figure 12: Load on Physalia vs. *agg* failure rate for a simulated 3-tier datacenter network.

6 Related Work

Physalia draws ideas from both distributed co-ordination systems and distributed databases. Distributed co-ordination systems, like Zookeeper [19], Chubby [9], Boxwood [38] and etcd [14], have the goal of providing a highly-available and strongly-consistent set of basic operations that make implementing larger distributed systems easier. Physalia's design approach is similar to some of these systems, being based on the state machine replication pattern popularized by the work of Schneider [49], Oki [40] and Lampson [37]. Physalia's key differences from these systems are its fine-grained consensus (millions of distributed state machines, rather than a single one), and infrastructure awareness. This makes Physalia more scalable and more resistant to network partitions, but also significantly more complex.

The problem of providing highly-available distributed storage in fallible datacenter networks faces similar challenges to global and large-scale systems like OceanStore [34] and Farsite [3], with emphasis on moving data close to its expected to improve availability and latency. While the design of Physalia predates the publication of Spanner [15] and CosmosDB, Physalia takes some similar design approaches with similar motivation.

Horizontal partitioning of databases is a long-established idea for both scaling and availability. Systems like Dynamo [18] and its derivatives dynamically move partitions, and rely on client behavior or stateless proxies for data discovery. Dynamic discovery of high-cardinality data, as addressed by Physalia's discovery cache and forwarding pointers, has been well explored by systems like Pastry [47] and Chord [52]. Optimizing data placement for throughput and latency is also a well-established technique (such as in Tao [8], and Dabek et al [16]), but these systems are not primarily concerned with availability during partitions, and do not consider blast radius.

Physalia's approach to infrastructure-aware placement reflects some techniques from software-defined networking (SDN) [21]. Another similarity with SDN (and earlier systems, like RCP [10]) is the emphasis on separating control and data planes, and allowing the data plane to consist of simple packet-forwarding elements. This reflects similar decisions to separate Physalia from the data plane of EBS, and the data- and control planes of Physalia itself.

Infrastructure awareness, an important part of Physalia's contribution, seems to be an under-explored area in the systems literature. Some systems (like SAUCR [4], and the model proposed by Chen et al [13]) are designed to change operating modes when infrastructure failures occur or request patterns change, but we are not aware of other database explicitly designed to include data placement based on network topology (beyond simple locality concerns).

7 Conclusion

Physalia is a classic consensus-based database which takes a novel approach to availability: it is aware of the topology and datacenter power and networking, as well as the location of the clients that are most likely to need each row, and uses data placement to reduce the probability of network partitions. This approach was validated using simulation, and the gains have been borne out by our experience running it in production at high scale across over 60 datacenter-scale deployments. Its design is also optimized to reduce blast radius, reducing the impact of any single node, software, or infrastructure failure.

While few applications have the same constraints that we faced, many emerging cloud patterns require strongly consistent access to local data. Having a highly-available strongly-consistent database as a basic primitive allows these systems to be simpler, more efficient, and offer better availability.

Acknowledgments

The authors would like to thank David R. Richardson, Madhu Parthasarathy, Divya Jain, Surya Dhoolam, and the Physalia team for their work in supporting, developing and refining Physalia. We would also like to thank Marc Levy, Allan Vermeulen and James Hamilton for reviewing the initial Physalia design. Allan Vermeulen's work on distributed systems testing deeply informed our approach.

References

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi:10.1109/IEEESTD.2008.4610935.

[2] Hussam Abu-Libdeh, Robbert van Renesse, and Ymir Vigfusson. Leveraging sharding in the design of scalable replication protocols. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery. URL: https://doi.org/10.1145/2523616.2523623, doi:10.1145/2523616.2523623.

[3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002. URL: http://doi.acm.org/10.1145/844128.844130, doi:10.1145/844128.844130.

[4] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Fault-tolerance, fast and slow: Exploiting failure asynchrony in distributed systems. In

13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 390–408, Carlsbad, CA, October 2018. USENIX Association. URL: https://www.usenix.org/conference/osdi18/presentation/alagappan.

[5] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 51–68, Carlsbad, CA, October 2018. USENIX Association. URL: https://www.usenix.org/conference/osdi18/presentation/alquraan.

[6] Peter Bailis and Kyle Kingsbury. The network is reliable. *Commun. ACM*, 57(9):48–55, September 2014. URL: http://doi.acm.org/10.1145/2643130, doi:10.1145/2643130.

[7] Philip A Bernstein and Sudipto Das. Rethinking eventual consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 923–928. ACM, 2013.

[8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.

[9] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1298455.1298487.

[10] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251203.1251205.

[11] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186. USENIX, 1999. URL: http://pmg.csail.mit.edu/papers/osdi99.pdf.

[12] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1281100.1281103, doi:10.1145/1281100.1281103.

[13] Ang Chen, Hanjun Xiao, Andreas Haeberlen, and Linh Thi Xuan Phan. Fault tolerance and the five-second rule. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/chen.

[14] Cloud Native Computing Foundation. etcd: A distributed, reliable key-value store for the most critical data of a distributed system, 2019. URL: https://etcd.io/.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[16] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a dht for low latency and high throughput. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, page 7, USA, 2004. USENIX Association.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013. URL: http://doi.acm.org/10.1145/2408776.2408794, doi:10.1145/2408776.2408794.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1294261.1294281, doi:10.1145/1294261.1294281.

[19] Tobias Distler, Frank Fischer, Rüdiger Kapitza, and Siqi Ling. Enhancing coordination in cloud infrastructures with an extendable coordination service. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, SDMCMM '12, pages 1:1–1:6, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2405186.2405187, doi:10.1145/2405186.2405187.

[20] John (JD) Douceur. The sybil attack. In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, January 2002. URL: https://www.microsoft.com/en-us/research/publication/the-sybil-attack/.

[21] MV Fernando, Paulo Esteves, Christian Esteve, et al. Software-defined networking: A comprehensive survey. *PROCEEDINGS OF THE IEEE*, 2015.

[22] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. URL: http://doi.acm.org/10.1145/564585.564601, doi:10.1145/564585.564601.

[23] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 15–28, New York, NY, USA, 2011. Association for Computing Machinery. URL: https://doi.org/10.1145/2043556.2043559, doi:10.1145/2043556.2043559.

[24] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991. URL: https://doi.org/10.1145/103162.103163, doi:10.1145/103162.103163.

[25] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM. URL: http://doi.acm.org/10.1145/74850.74870, doi:10.1145/74850.74870.

[26] Jim Gray. A census of tandem system availability between 1985 and 1990. Technical Report TR-90.1, HP Labs, 1990.

[27] Shay Gueron and Yehuda Lindell. Gcm-siv: Full nonce misuse-resistant authenticated encryption at under one cycle per byte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 109–119. ACM, 2015.

[28] Neil J. Gunther. A general theory of computational scalability based on rational functions. *CoRR*, abs/0808.1431, 2008. URL: http://arxiv.org/abs/0808.1431, arXiv:0808.1431.

[29] James Lee Hafner and KK Rao. Notes on reliability models for non-mds erasure codes. Technical report, IBM Research Division, 2006.

[30] James Hamilton. Aws re:invent 2016: Tuesday night live with james hamilton. URL: https://www.youtube.com/watch?v=AyOAjFNPAbA.

[31] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. URL: http://doi.acm.org/10.1145/78969.78972, doi:10.1145/78969.78972.

[32] Yury Izrailevsky and Ariel Tseitlin. The Netflix Simian Army, 2011. URL: https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116.

[33] Kyle Kingsbury. Jepsen, 2019. URL: https://jepsen.io/.

[34] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 190–201, New York, NY, USA, 2000. ACM. URL: http://doi.acm.org/10.1145/378993.379239, doi:10.1145/378993.379239.

[35] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[36] Leslie Lamport. Who builds a house without drawing blueprints? *Communications of the ACM*, 58(4):38–41, 2015.

[37] Butler W. Lampson. How to build a highly available system using consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG '96, pages 1–17, London, UK, 1996. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645953.675640.

[38] John MacCormick, Nick Murphy, Marc Najork, Chandu Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating System Design and Implementation (OSDI)*, pages 105–120. USENIX, December 2004.

[39] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[40] Brian M. Oki. *Viewstamped Replication for Highly Available Distributed Systems*. PhD thesis, MIT, 8 1988.

[41] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM. URL: http://doi.acm.org/10.1145/62546.62549, doi:10.1145/62546.62549.

[42] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2643634.2643666.

[43] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM. URL: http://doi.acm.org/10.1145/50202.50214, doi:10.1145/50202.50214.

[44] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1267903.1267905.

[45] J. Postel. Dod standard transmission control protocol. RFC 761, RFC Editor, January 1980.

[46] David R. Richardson. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. URL: https://aws.amazon.com/message/65648/.

[47] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646591.697650.

[48] Nuno Santos and Andre Schiper. Optimizing Paxos with batching and pipelining. *Theoretical Computer Science*, 496:170–183, 2013. doi:10.1016/j.tcs.2012.10.002.

[49] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. URL: http://doi.acm.org/10.1145/98163.98167, doi:10.1145/98163.98167.

[50] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance-computing systems. Technical Report CMU-PDL-05-112, Carnegie Mellon University, 2005.

[51] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, Berkeley, CA, USA, 2007. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1267903.1267904.

[52] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM. URL: http://doi.acm.org/10.1145/383059.383071, doi:10.1145/383059.383071.

[53] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149. IEEE, 1994.

[54] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251254.1251261.

[55] Peter Vosshall. Aws re:invent 2018: How aws minimizes the blast radius of failures. URL: https://www.youtube.com/watch?v=swQbA4zub20.

[56] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, pages 54–66, London, UK, UK, 1999. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646704.702012.

[57] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, Broomfield, CO, 2014. USENIX Association. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan.

Diamond-Miner: Comprehensive Discovery of the Internet's Topology Diamonds

Kevin Vermeulen
Sorbonne Université

Justin P. Rohrer
Naval Postgraduate School

Robert Beverly
Naval Postgraduate School

Olivier Fourmaux
Sorbonne Université

Timur Friedman
Sorbonne Université

Abstract

Despite the well-known existence of load-balanced forwarding paths in the Internet, current active topology Internet-wide mapping efforts are multipath agnostic – largely because of the probing volume and time required for existing multipath discovery techniques. This paper introduces D-Miner, a system that marries previous work on high-speed probing with multipath discovery to make Internet-wide topology mapping, inclusive of load-balanced paths, feasible. We deploy D-Miner and collect multiple IPv4 interface-level topology snapshots, where we find >64% more edges, and significantly more complex topologies relative to existing systems. We further scrutinize topological changes between snapshots and attribute forwarding differences not to routing or policy changes, but to load balancer "remapping" events. We precisely categorize remapping events and find that they are a much more frequent contributor of path changes than previously recognized. By making D-Miner and our collected Internet-wide topologies publicly available, we hope to help facilitate better understanding of the Internet's true structure and resilience.

1 Introduction

An important component of today's Internet is multipath routing [18, 29, 45], where traffic to a destination network is *load-balanced* to support higher capacities and provide redundancy. Prior work developed active measurement techniques to discover multipath routing [17], while showing that multipath topologies can be quite complex [45]. However, high probing loads and runtime have been impediments to their widespread uptake. As a result, today's IP and router topology snapshots, e.g., [21, 41], incompletely represent the true, multipath, complex forwarding topology. Indeed, our measurements discover >2.7M more edges (64%) in the topology as compared to current state-of-the-art, and significantly more complex topologies than previously reported, including >5k edges in a single provider's topology corresponding to a single /24 destination prefix that they advertise.

The Internet measurement community has long aimed to capture a complete topology of the Internet, especially as it has become clear that partial topologies can lead to faulty conclusions about the network's properties [15, 24, 46]. Obtaining not just accurate, but complete topologies is crucial to understanding the network's resilience to outages and attacks [35, 43], as well as aiding in the conception of applications ranging from content distribution to network security [48]. Further, these topologies play a vital supporting role in other measurement inferences, for instance determining AS relationships, mapping inter-domain congestion [32], and geolocation [31] to name a few.

We developed D-Miner to gather Internet-wide multipath topology maps. D-Miner is a system that marries two recent advancements in active topology discovery: i) high-speed randomized probing techniques [19] and ii) multipath detection algorithms [17]. At its core, D-Miner rapidly sends probes in a randomly permuted order to avoid overloading any single path, and iteratively completes its view of the network. Whereas prior multipath discovery approaches perform a stateful breadth-first search path exploration to attain confidence in the degree of load balancing along the path, D-Miner decouples probing from statistical inference thereby enabling probe randomization and high-rate probing, while amortizing total probing cost. D-Miner maintains a set of "unresolved" nodes – nodes for which the degree of load balancing is uncertain – and proceeds in rounds until the topology is, statistically, complete (§3). Having implemented D-Miner, we evaluate its performance and overhead (§4). Together, these techniques enable, for the first time, *scalable* multipath topology discovery and Internet-wide mapping.

Using complete Internet-wide snapshots collected using D-Miner, we scrutinize dynamics and topological changes. We find that many "routing changes" are instead due to "load balancer remapping" events – where the load balancer changes its current mapping between flow identifiers and paths. We precisely categorize these events and measure their prevalence, finding that they are a much larger and frequent contributor of path changes than previously recognized (§5).

Finally, we deployed D-Miner and collected multiple topology snapshots over a one-week period in August, 2019. From these snapshots, we characterize the prevalence, size, extent, and location of load-balancing on the modern Internet (§6). Our contributions thus include:

1. Development and evaluation of D-Miner, a novel scalable active multipath topology discovery system.
2. Deployment of D-Miner to gather the first Internet-wide IP-level topology snapshots inclusive of load balancing.
3. Detailed characterization of Internet dynamics, including a taxonomy of load balancer remapping events and their extent and prevalence.
4. Public release of D-Miner's code and survey results [4].

2 Background and related work

We first review different types of load balancing commonly found in the Internet. Then, we provide an overview of the Multipath Detection Algorithm (MDA) [44], the current state-of-the-art technique for actively discovering load balancing, and Yarrp, a method for high-speed topology discovery. We finish this section by describing other related work.

2.1 Load balancing

Load balancing is used to increase aggregate network capacity and provide redundancy and resilience to failures. Two types of load balancing are configurable on routers [3, 6, 12]: deterministic and non-deterministic. When a packet arrives on a router configured with deterministic load balancing (i.e., per-flow load balancing), and multiple equal-cost routing paths are available to the packet's destination, the router chooses a path by computing a hash over the packet's header fields [23]. This set of fields used to compute the hash is called the flow identifier, and typically includes either the source and destination addresses (per-destination) or the source and destination addresses and ports (per-flow). Two packets belonging to the same flow are thus sent over the same path, and this helps the performance of transport protocols that react to delayed or out-of-order packets, as well as enabling middleboxes to have visibility into all the packets of a flow. Herein, we use the terms "flow identifier" and "flow" interchangeably.

Non-deterministic is also known as per-packet load balancing. In this configuration, when a packet arrives at a router with multiple equal-cost paths to the destination, the router selects among the paths in a round robin fashion.

Our Internet scale survey in §6 confirms two previous results of Augustin et al. [18] and Vermeulen et al. [45]: (1) load balancing is prevalent in the network, as 64.7% of our traces from a source to any /24 prefix contained at least one load balancing router (branching point); and (2) non-deterministic load balancing is rare, with only 1.9% of the branching points identified as implementing this behavior.

2.2 MDA

The Traceroute tool [30] sends probe packets to find forwarding paths. It exploits the IPv4 time-to-live (TTL) header field to induce routers along a forwarding path to send ICMP error messages, thereby revealing the router's interface addresses. The original Traceroute design did not foresee the later emergence of load-balanced paths in the Internet, and it gives incomplete and incorrect results in the face of load balancing [16]. Paris Traceroute [16] was developed specifically to accurately reveal a path through a per-flow load-balanced network. Paris Traceroute ensures that all probe packets, across different TTLs, have consistent flow identifiers, thereby ensuring that all measurement packets take the same path in a load-balanced network. However, in its basic implementation, Paris Traceroute reveals only a single path to the destination.

The Multipath Detection Algorithm (MDA) stochastically varies Paris Traceroute's flow identifiers in an attempt to enumerate all paths to a destination. For a given vertex with k known outgoing load-balanced edges, the number of probes with randomly selected flow IDs needed to verify that it has no more than k edges is denoted n_k, and is termed a "stopping point" [44]. For example, when 1, 10, or 100 outgoing edges have already been identified, $n_1 = 6$, $n_{10} = 57$, or $n_{100} = 757$ probes are, respectively, required in order to ensure a no more than 0.05 probability to fail to enumerate all outgoing edges.

Unfortunately, the stateful nature of the MDA and its reliance on establishing confidence in the behavior of each potential branching point along the path in a sequential manner are hinderences to its use for Internet-wide topology studies.

Note, the MDA technique has previously been validated [18, 44]. From this perspective, if D-Miner achieves the same level of statistical guarantees that the MDA provides, it validates D-Miner as well. §4 shows that D-Miner fulfills this condition.

2.3 Yarrp

Yarrp [19, 20] introduced the notion of high-speed topology probing via stateless operation, and random permutation of targets and TTLs. Whereas previous route tracing techniques, e.g., [34], iteratively probe TTLs toward each destination, Yarrp randomizes its probing and decouples probing from topology reconstruction. This randomization avoids overloading particular paths or routers, thereby permitting higher probing rates. Further, Yarrp encodes all of the necessary state, e.g., originating TTL and time, into probes such that the quoted replies permit state to be reconstituted. In this fashion, Yarrp demonstrated the ability to perform Internet-wide route tracing at more than 100k pps.

2.4 Other related work

For more than two decades now, Internet mapping has been an active area of research. It allows researchers to better un-

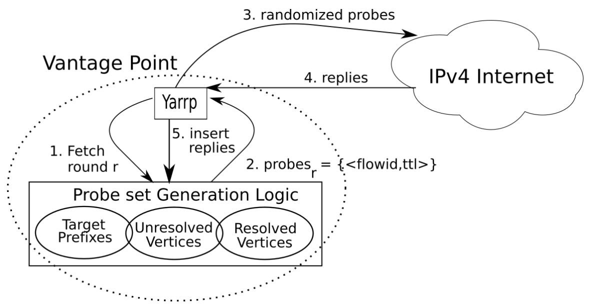


Figure 1: D-Miner high-level conceptual overview

derstand the structure of the Internet [22, 28] and to design better protocols [46]. Today, one can either obtain an Internet snapshot of the whole Internet without load-balanced paths, or a reduced snapshot of the Internet with load-balanced paths. Mapping systems such as CAIDA's Ark [24] perform continuous surveys from hundreds of vantage points by launching Paris Traceroute measurements to one random address in every globally advertised /24 prefix. A complete set of measurements towards all of the /24 prefixes is called a cycle. Because the address in the /24 is randomly chosen, aggregating results from multiple cycles reveal some load balancing. However, Ark is not explicitly designed to reveal load balancing, does not send enough probes to find all load balancing (particularly when chained), and does not provide any confidence bounds on discovery. Yarrp [19], described in §2.3, performs Internet scale surveys at high speed, but is also not capable of revealing the load-balanced paths. D-Miner enables researchers, for the first time, to obtain snapshots the entire Internet inclusive of all load-balanced paths.

This new and more complete view of the topology allows D-Miner to expand the results on load balancing characterization at Internet scale. Augustin et al. made the first study of load-balanced paths a decade ago, finding topologies having up to 16 such paths [18]. More recently, Vermeulen et al. have shown that per-flow load-balanced paths have become more complex, by finding topologies with up to 92 interfaces at the same TTL [45]. In both of those studies, the set of destination prefixes was a subset of the entire Internet, with, respectively, $\sim$120k and $\sim$350k targets. In contrast our survey contains traces to $\sim$14.4M /24 destination prefixes.

D-Miner allows us to provide new insights into Internet dynamics induced by load balancing. In §5.2, we show that a "routing change" is more nuanced that previously understood, and that such changes can be due in part to the remapping behavior of production load balancers. Paxson's canonical work on Internet dynamics [39] studied *persistence* and *prevalence* of routes. Given a source/destination pair, persistence characterizes the number of different routes observed across time between this pair. The prevalence of a route defines if, within the set of routes that have been observed, one is more dominant than another. The main result was that Internet routes were globally stable across time. Note that this work was performed two decades ago and did not take load balancing into account. Almost a decade ago, Cunha et al. reappraised Paxon's work in light of load balancing [26], and found that Paxson's results still held. They also stated that load balancing remapping is infrequent. We show that it is a widespread phenomenon in today's Internet.

More recently Cunha et al. developed DTrack [27], a system that maintains an inferred topology and attempts to detect and predict path changes, although such prediction is difficult. Our work helps provide insight into potential root causes of this prediction difficulty.

3 Algorithm

D-Miner is designed to capture Internet topology snapshots inclusive of all load-balanced paths. At its heart, D-Miner uses Yarrp's randomized and stateless probing to achieve high probing rates. To this, it adds probe set generation logic that keeps track, on a per-node basis, of whether all outbound load-balanced edges have been discovered with high probability. The logic guides Yarrp through multiple rounds until the full discovery criterion has been satisfied for almost all nodes.

See Figure 1 for a high-level schematic of D-Miner. Yarrp and the probe set generation logic are deployed at a single vantage point from which Yarrp probes a set of target prefixes in the IPv4 Internet. D-Miner proceeds in rounds, maintaining sets of "resolved" and "unresolved" vertices. Resolved vertices correspond to nodes where all outgoing load balanced links have been discovered with high probability toward the set of target prefixes. Conversely, unresolved vertices require further probing to ascertain whether any load balanced edges emergent from a node have not yet been discovered.

D-Miner's main steps are: (1) Yarrp requests the set of probes for round *r*; (2) the probe set generation logic returns the `<flow ID,TTL>` pairs that correspond to the current set of unresolved vertices; (3) these pairs are randomized and probes are sent at high speed using the Yarrp technique; (4) as replies return, they are processed by Yarrp; and (5) they are used to update the set of known nodes and each node's state as either an unresolved or a resolved vertex. Rounds continue until 99% of the target prefixes have been resolved.

Whereas Yarrp is totally stateless, D-Miner requires state to be retained from round to round. A key challenge is to manage that state in a manner that does not diminish Yarrp's performance. Our solution is discussed in §3.2.

3.1 Bootstrapping D-Miner

Since D-Miner is guided by the set of unresolved vertices, it requires a boostrapping round to seed the set. This round is a slightly modified version of a classic Yarrp snapshot of the IPv4 Internet. Whereas Yarrp sends one probe packet per TTL to each /24 prefix, with the exception of the private and reserved IPv4 prefixes defined by RFC 6890 [25], D-Miner sends $n_1 = 6$ probes per /24, each with a different flow identifier. This number comes from the MDA stopping condition for 0.05 failure probability, described in §2.2, that there is just a single node at a given TTL when probing towards a given destination. The six flow identifiers correspond to the six first destinations in the /24. The /24 granularity corresponds to the commonly accepted longest BGP prefix [42].

In the classic MDA, the n_1 packets all have a common destination, however D-Miner varies the flow identifier by varying the destination within the target prefix. This allows it to find per-destination-prefix load balanced paths in addition to the per-flow load balanced paths that classic MDA finds.

As stated in §2.3, Yarrp uses a pseudo random permutation of 32 bits to determine the parameters for each successive probe: the first 24 bits determine the /24 destination prefix and the first 5 bits of the remaining byte determine the TTL. This leaves 3 bits, which is sufficient for D-Miner to select $n_1 = 6$ different addresses within the destination prefix.

3.2 Maintaining State

Yarrp encodes the originating TTL, timestamp, and checksum in the probe header fields. These values are visible in the quotations that arrive in the probe replies, allowing Yarrp to reconstruct the probe that generated a particular ICMP without maintaining any internal state. While each individual Yarrp probing round is stateless, D-Miner must maintain state from round to round in to keep track of which vertices have been resolved and which ones remain unresolved. To this end, D-Miner extracts the following data from each reply: the original probe's source and destination IP addresses, port numbers, and original TTL; and the reply's source IP address and ICMP type and code.

So that D-Miner could obtain rapid results for complex queries on tables of billions of rows, we sought a database system that is optimized for online analytical processing, settling on ClickHouse [2]. The data from each reply is inserted and ordered by: source IP address, /24 destination prefix, destination IP address, TTL, source port number, and destination port number. ClickHouse is highly parallelized and its `groupArray` features make the algorithm calculations described in §3.3 and the analyses of §4, §5, and §6 tractable.

3.3 Subsequent probing round computation

Once the replies have been inserted into the database, we query it to generate the next round of probes. Our goal, conceptually, is to calculate the set of additional probes with new flow identifiers required to meet the remaining statistical guarantees for each /24 prefix, given the current knowledge of the topology. Mathematically, the next round probes is the minimal expected set of probes needed to reach the statistical guarantees for each branching point, grouped by /24 prefix.

3.3.1 Reducing MDA statefulness

This section describes our algorithm to generate a new batch of probes given a topology and the set of probes already sent.

Let us fix a source and a /24 destination prefix. We present an example in Figure 2 to help provide intuition. This topology illustrates a possible result after each of three hypothetical rounds of probing. Each link is annotated with the number of probes that expired at the ingress interface of the subsequent node. At round 1, D-Miner sent $n_1 = 6$ probe packets per TTL, each with varying destinations in the destination prefix to vary the flow identifier. The value of n_1 is determined by the desired failure probability to find all the successors of a branching point. In this work we set $n_1 = 6$, corresponding to a failure probability of 0.05, which is the default value used by the MDA implementation in previous work [18, 27].

Recall, after sending 6 probes and only discovering a single successor, we have a probability of 0.95 that there is indeed only a single successor ($n_1 = 6$), while we must send 11 probes to achieve the same probability that there are only two successors ($n_2 = 11$). To understand how we compute the next batch of probes, we introduce the following notation:

Let us fix the TTL to h.

Let R_h be the set of nodes discovered at TTL h that have not been resolved yet.

Let D_h be the probability distribution of nodes responding for TTL h after the current probing round. For example, in Figure 2 after the first round of six probes per TTL, $D_2 = \left\{\mathrm{v}_2 = \frac{4}{6}, \mathrm{v}_3 = \frac{2}{6}\right\}$.

Let k_v be the number of successors for node v.

Let t_h be the number of probes already sent at TTL h.

Let n_k be the stopping point described in §2.2.

Proposition 1. *Given h, R_h and D_h, the minimal expected number of probes needed to reach MDA statistical guarantees for all the elements of R_h is:*

$$\max_{v \in R_h}\left(\frac{n_{k_v}}{D_h(v)} - t_h\right) \qquad \text{At TTL } h$$

$$\max_{v \in R_h}\left(\frac{n_{k_v}}{D_h(v)} - t_{h+1}\right) \qquad \text{At TTL } h+1$$

Proof. Let $v \in R_h$. The MDA hypothesis, which is that v might have a $(k_v+1)^{\text{th}}$ successor tells us that we need to send n_{k_v} probe packets with TTL h that first reach v, and then send n_{k_v} with the same flow identifiers to TTL $h+1$. Let X_v be the random variable representing the number of probes that reach node v given D_h. Our objective is to find the minimum number of probes N such that:

$$\forall v, E[X_v] = D_h(v)N >= n_{k_v} \qquad (1)$$

For this condition to hold, we must set N at minimum to:

$$N = \max_{v \in R_h} \frac{n_{k_v}}{D_h(v)}$$

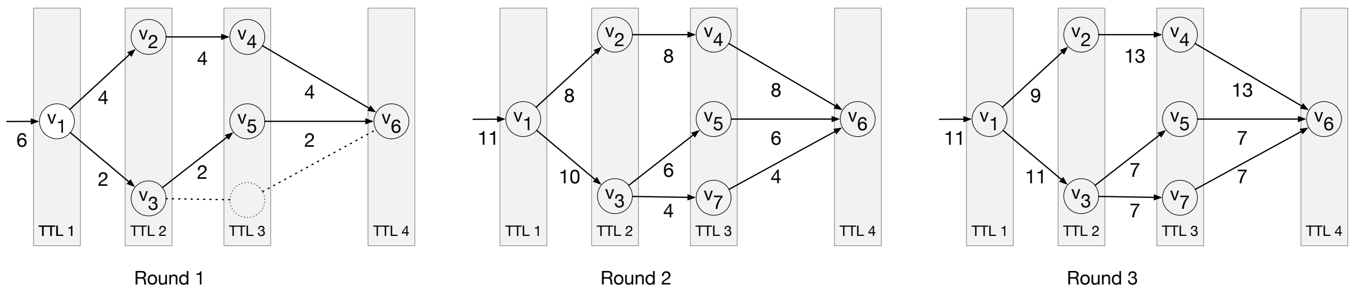


Figure 2: Example topology resolved by three rounds of D-Miner probing. Link annotations represent number of probes expiring at ingress interface of subsequent node.

We have:

$$\forall v, E[X_v] = D_h(v) \max_{v' \in R_h} \frac{n_{k_{v'}}}{D_h(v')} >= D_h(v) \frac{n_{k_v}}{D_h(v)} = n_{k_v} \quad (2)$$

We just subtract the number t of probes that we have already sent to TTL h, and this concludes the proof. □

Every TTL h generates a number of additional probes for TTL h and TTL $h+1$. For each TTL h, we therefore have two possible values: the one generated by additional probes for TTL $h-1$ and the one generated by additional probes for TTL h. So that the condition given in Eq. 1 holds for every node of the topology, we choose the maximum between these two values (rounding fractional values to integers).

Let us perform the numerical application on the topology of Figure 2. After round 1, we have discovered the topology on the left side of the figure. At TTL1, Node v_1 has two successors, $D_1(v_1) = 1$. 6 probes have been sent to TTL 1 and 6 probes have been sent to TTL 2. Proposition 1 brings $n_2 - 6 = 5$ additional probes for TTL 1 and 2. At TTL 2, Node v_2 has one successor, and $D_2(v_2) = \frac{2}{3}$. Node v_3 has one successor, and $D_2(v_3) = \frac{1}{3}$. 6 probes have been sent to TTL 2 and 6 probes have been sent to TTL 3. Proposition 1 brings $3n_1 - 6 = 12$ additional probes for TTL 2 and 3. At TTL 3, notice that v_4 and v_5 are similar to v_2 and v_3, so that Proposition 1 brings $3n_1 - 6 = 12$ additional probes for TTL 3 and TTL 4. For each TTL, we take the maximum number of probes between TTL and TTL-1. At the end, we have to send for the second round: 5 probes at TTL 1, 12 at TTL 2, 12 at TTL 3 and 12 at TTL 4.

The figure in the center shows the state of the topology after round 2. At TTL 1, we see that v_1 has been resolved because it has two successors and $11 = n_2$ flows pass through it. At TTL 2, v_2 has also been resolved because it has one successor and $8 \geq n_1$ flows have been sent through it. v_3 is not solved, because now it has two successors so it needs $n_2 = 11$ flows that pass through it. We have $D_2(v_3) = \frac{10}{18}$ and 18 probes have been sent to TTL 2 and 3. Proposition 1 brings $\frac{18}{10}n_2 - 18 = \lceil 1.8 \rceil = 2$ additional probes for TTL 2 and 3. At TTL 3, we see that v_4 and v_5 have been resolved, but v_7 has not. We have $D_3(v_7) = \frac{4}{18}$ and 18 probes have been sent to TTL 3 and 4. Proposition 1 brings $\frac{4}{18}n_1 - 18 = 9$ additional probes for TTL 3 and 4. For each TTL, we take take the maximum number of probes between TTL and TTL-1. At the end, we have to send for the third round: 0 probe at TTL 1, 2 at TTL 2, 9 at TTL 3 and 9 at TTL 4.

The figure on the right shows the state of the topology after round 3. We see that now v_3 and v_7 have been resolved, so that we consider the entire topology as resolved.

3.3.2 Varying the flow identifier

For each destination prefix and TTL where additional probes are needed, we select new flow identifiers by incrementing the last destination in the prefix used in the previous round. In the example of Figure 2, suppose that our prefix is X.Y.Z.0/24. After round 1, the first 6 IPs of the prefix have already been used as probe destinations, so we would send 5 more probes at TTL 1 from X.Y.Z.7 to X.Y.Z.11, 12 more probes at TTL 2 from X.Y.Z.7 to X.Y.Z.18, etc. If there are no more destinations available in the prefix, we change the destination port to vary the flow identifier.

3.4 Randomizing the probes

Randomizing the probes is done per-flow. For each prefix which needs additional probes sent, we group the additional probes by flow. In the example of Figure 2, after round 1, we send X.Y.Z.7 at TTL 1, TTL 2, and TTL 3, and TTL 4. All of these are packed together so that we ensure that probes with the same flow identifier are sent in a very short time window. This avoids the inference of false links due to potential routing changes during the probing time. Nevertheless, it is possible that X.Y.Z.8 probes are sent at a separate time from X.Y.Z.7. In this way we retain one of the benefits of Yarrp's randomization, to minimize potential ICMP rate limiting.

3.5 Per-Packet load balancing

As described in §3.1, the first round of probes allows us to discover if there is one or more load-balancers on the path from the vantage point to the destination prefix. However, sending only one packet per flow identifier does not reveal the nature of the load balancing, i.e., deterministic (per-destination or per-flow) or non deterministic (per packet). For per-packet load balancers, the links between interfaces of a traceroute can not be reliably inferred. Because we want to be able to flag per packet load balancers, D-Miner sends two back-to-back probes per flow instead of one until it reaches a defined threshold probability that the branching point is not a per packet load balancer. If we get two different responses for flows with the same flow identifier, the branching point is flagged as a per packet load balancer and the edges discovered after it are ignored in the results. Mathematically, an upper bound of the probability to miss that a load balancer is actually a per packet load balancer is the probability that all the pairs of probes with the same flow identifiers passing through it get the same reply IP. Suppose there is a branching point with k branches. Then the probability that all these pairs of probes get the same response is then $\frac{1}{k^p}$ where p is the number of pairs of probes sent going through this branching point. We set the threshold probability in D-Miner to 0.95, as previously done in [18].

4 Evaluation

Over one third of the edges in the Internet's topology are not being revealed by current state-of-the-art methods, as §4.4 describes. D-Miner, run from from six PlanetLab Europe vantage points, discovered more than 7.1 million edges during a week of August 2019, a time during which Ark, probing from its 112 VPs, found ~4.4 million edges and Yarrp, probing from the same PLE VPs as D-Miner, found ~2.5 million. Although this additional coverage comes at the cost of 2-4 times as many probes compared to these existing approaches (§4.4.1), we show that this volume is required due to forwarding path dynamics.

In addition to evaluating node and edge discovery, we evaluate D-Miner's algorithmic and system properties. D-Miner's round-iterative design engenders 14% higher overhead than would be incurred sending traditional source-to-destination style MDA Paris Traceroutes from the same vantage point, as §4.2 shows. This is the cost associated with D-Miner adopting a stateless and randomized (Yarrp-based) design, in exchange for the advantages conferred by this design in terms of probing speed. §4.3 evaluates D-Miner's running time. First, however, §4.1 describes the datasets we collected, and shows that 10 rounds suffice to reach statistical guarantees for 99% of the /24 destination prefixes.

4.1 Dataset

Our evaluation dataset includes three D-Miner snapshots, each consisting of 10 rounds. These snapshots were collected over a one-week period, from 6-13 August 2019, using a single vantage point at our university, which enjoys a 1 GB link to the Internet. Each snapshot was collected using a probing rate of 100,000 pps; this rate was capped out of respect for network and service provider policy concerns; D-Miner is capable of probing much faster (>800,000 pps observed).

In addition, for the topology discovery section, we have deployed D-Miner on six PlanetLab Europe (PLE) [7] nodes located in Europe and run it at a lower rate, 10,000 pps, during the same week. We used UDP probes as these have been shown to discover more links than other protocols [36]. A web page hosted at the IP address of our vantage point described the experiment and provided instructions for opting out; we did not receive any such requests during our measurements.

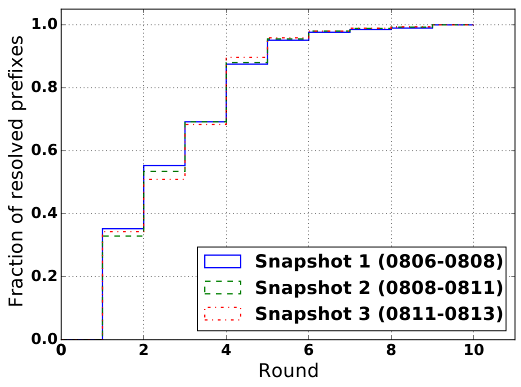


Figure 3: CDF of /24 IPv4 prefixes meeting their statistical guarantees, over three 10-round snapshots (dates in parens)

Table 1: Probing Overhead of D-Miner versus MDA.

Overhead Factor	**Snapshot 1** (Aug 6-8)	**Snapshot 2** (Aug 8-11)	**Snapshot 3** (Aug 11-13)
Loss (I)	481,359,024	448,428,279	569,173,354
Reached destination (II)	166,941,456	165,227,106	163,821,372
Last round (III)	54,023,123	46,324,355	50,012,378
Total Overhead [pkts]	**702,323,603**	**659,979,740**	**783,007,104**
D-Miner Sent [pkts]	6,976,307,081	6,569,819,008	6,598,837,985
MDA Sent [pkts]	6,273,983,478	5,909,839,268	5,815,830,881
D-Miner Overhead [%]	**12**	**12**	**14**

Figure 3 illustrates how 10 rounds of probing suffices to achieve statistical guarantees toward more than 99% of the IPv4 /24 destination prefixes. The mean portion of resolved prefixes over the three 10-round snapshots is 99.6%.

4.2 Probing overhead

Intuitively, we might expect D-Miner to have lower overhead in order to achieve scale. However, the stateless high-rate probing required comes at the cost of additional total probing. The probing overhead generated by D-Miner compared to sequentially running the traditional MDA to all of the /24 prefixes depends on three factors: (I) potential loss induced

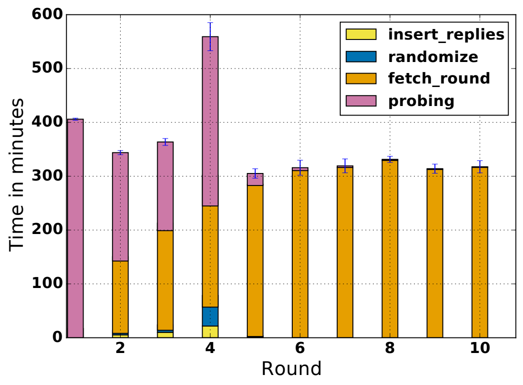


Figure 4: Time spent in each step of D-Miner for each round, averaged across three snapshots.

by a high probing rate; (II) the TTL at which the MDA would have stopped because of reaching the destination; and (III) sending more probes in the last round than required to reach the statistical guarantees. Table 1 quantifies each of these overhead factors across our three snapshots, where we find a maximum overhead of 14%.

Note that we conservatively compute the loss due to high probing rates. If for a given (destination prefix, TTL) pair, if at least one of the probes receives one response, we count all the probes sent with the same (destination prefix, TTL) pair as losses if they do not produce a response. Conversely, if no responses at all are received for this (destination prefix, TTL) pair, i.e., this hop is "anonymous"[1], we do not count these probes as lost.

To compute the TTL at which the MDA would have stopped, we find the minimum TTL for which all probes' reply IPs equal their destination IP. If we never receive a reply from the destination, we assume that MDA would act like a default linux traceroute, i.e sending probes until it reaches the maximum TTL (30).

And finally, to compute the last round overhead, we simulate for each of the destination prefixes a run of the MDA with the same flow identifiers and compute the actual probe at which it stops due to having reached the statistical guarantees.

4.3 Run time

The server used for the computation was the same as the one we used for probing, provisioned with 16 cores and 187 GB of RAM. Figure 4 shows the stacked error bars of the time spent across each round in each routine. The sum over the rounds indicates that a snapshot of 10 rounds takes an average of 3,713 minutes (about 2 days and 14 hours) to complete.

We distinguish two phases in our system: Until round 4, the `probing` routine consumes a significant portion of the total round time, however, in rounds 5–10, almost all the time is spent in the `fetch_round` routine. Note the relationship

[1]Represented as a * in traditional traceroute output [9]

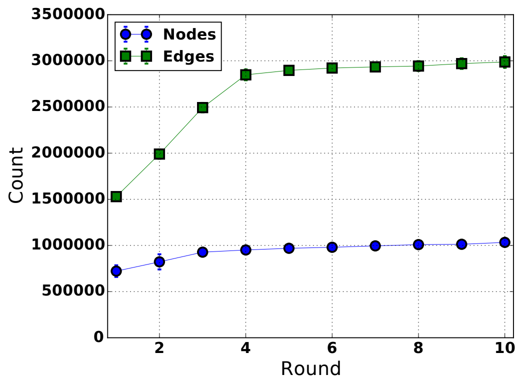


Figure 5: Per-round node / edge discovery for three snapshots.

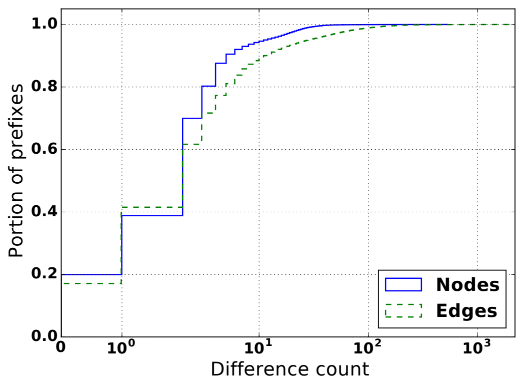


Figure 6: CDF of the minimum difference of number of nodes and edges per prefix.

between time (Figure 4) and the fraction of resolved prefixes (Figure 3). While the amount of probing time required in rounds 6-10 is relatively inexpensive as compared to the earlier rounds, we see that the algorithm is primarily optimizing at this point, with >90% of the prefixes resolved by the fifth round. Conversely, only ∼50% of the prefixes are resolved after the third round, demonstrating that the runtime cost of probing in the early rounds is required.

Notice the evolution of the time spent in the `fetch_round` routine: The time needed to compute the next round probes increases with the size of the table of our database in rounds 1–4. Then, from round 5 on, the reduction in time spent `probing` indicates that far fewer probes are sent, and consequently the table does not grow as much as during the earlier rounds. Still, the time of `fetch_round` remains constant; one must recompute the state of each branching point at each round to compute the next one.

4.4 Topology discovery

Finally, we consider the topology discovery results themselves. Note we ignore responsive target addresses (since we are only interested in the routed topology) as well as any edges connecting the destination nodes.

Table 2: Topology discovery

	Nodes				Edges			
	Snapshot 1 (Aug 6-8)	Snapshot 2 (Aug 8-11)	Snapshot 3 (Aug 11-13)	Agg.	Snapshot 1 (Aug 6-8)	Snapshot 2 (Aug 8-11)	Snapshot 3 (Aug 11-13)	Agg.
D-Miner	1,057,832	1,017,389	1,024,178	1,373,149	2,906,204	2,997,581	3,059,770	4,600,689
Yarrp	545,536	492,694	516,363	632,405	977,201	898,607	916,196	1,279,171
Multi-VP Ark	1,432,604	1,635,779	1,343,009	1,923,038	3,044,747	3,472,120	2,994,190	4,365,515
Multi-VP Yarrp	802,891				2,483,816			
Multi-VP D-Miner	1,613,301				7,143,490			

Figure 5 shows the cumulative discovery, with error bars, of each round across the three D-Miner snapshots. We find that both nodes and edges have similar behavior with two discovery phases, with an initial high discovery-per-round phase until round 3 (nodes) and 4 (edges) followed by a refinement phase from round 4 (nodes) and 5 (edges) to round 10. Note also that the number of nodes and edges varies little across the three snapshots, however, this does not mean that they discover the same set of nodes and edges as we will discuss.

It is challenging to make direct comparisons previous topology data sets – not only is the network dynamic, but the results are also significantly influenced by differing vantage point(s). Instead, we seek to make a reasonable comparison of D-Miner with existing Yarrp and Ark systems.

To compare with Yarrp, we extract from our D-Miner snapshots results given by selecting only one destination per /24 prefix in the first round. To compare against Ark, we obtain all topology traces from all 112 vantage points corresponding to the date range of our snapshots (this includes 19 "cycles" that were performed during the week). We aggregate the topology found from all cycles during the D-Miner snapshot collection to compare findings from the same time period.

Table 2 gives the comparative topology results. The "Agg." column shows the aggregated results over the three snapshots. The two last lines of Table 2 show the aggregated topology discovery results across the six PLE nodes for Yarrp and D-Miner over the same week covered by the three snapshots.

The multiple vantage point results in this table are significant. With 6 vantage points, D-Miner discovers >7M edges and approximately 1.6M nodes. To verify that this difference is primarily due to load balancing, we look at how standard Yarrp performs when used from these 6 vantage points. We see that D-Miner discovers two times more nodes and almost three times more edges than Yarrp.

Interestingly, we see that D-Miner from a single vantage point still benefits from snapshot aggregation. This means that there are non-negligible variations in the discovery of the three snapshots. Figure 6 shows the minimum per-prefix difference of number of nodes and edges discovered between the three snapshots over the 14,461,947 prefixes that we probed. As an example, for a prefix, if snapshot 1 discovers 20 nodes and 40 edges, snapshot 2 discovers 20 nodes and 34 edges, and snapshot 3 discovers 20 nodes and 36 edges, the minimum difference for nodes is 0 and 4 for edges. We observe two results: Less than 20% of the prefixes discover the same number of nodes and edges for the three snapshots, and 88.3% of the prefixes have a variation of less than 10 edges.

We believe that these variations are related to load balancing remapping (see §5.2). Indeed, the difference is not attributable to high probing rate induced loss, as Table 1 has shown than fewer than 10% of the probes were lost due to rate across the three snapshots.

Please note that we do not claim a comprehensive comparison with Ark discovery. The two systems have intrinsic differences that would not allow us to state conclusively why D-Miner or Ark would discover more than the other system. For example: (1) Ark uses ICMP probes, which has been demonstrated by Luckie et al. in [36] to discover fewer links than UDP. (2) The two systems' vantage points are not located at the same points in the network. Therefore, Ark could miss load balanced paths that are in a region of the Internet that is not accessible from its vantage points.

4.4.1 Probes sent

In total, we compute that Ark sent 5,935,460,660 probes. From Table 1 we compute that D-Miner from one vantage point sent 20,144,964,074 probes. And finally, we compute that the multiple vantage points version of D-Miner has sent 13,192,962,692 probes in total across the 6 PLE nodes.

These numbers give an idea of the overhead necessary to discover the load balanced paths. We show in the next section that reducing this number is hard because of the dynamics.

4.5 Validation on ground truth

To complement the formal and experimental analysis of our system, we solicited ground truth from operators. Of 380 interfaces with multipath edges discovered within Internet Initiative Japan (IIJ), they validated that 51 interfaces belonging to NTT were on PPPoE routers performing ECMP to IIJ. We further learned that the remaining 329 interfaces are performing ECMP inside IIJ's network.

In addition to direct correspondence with IIJ, we developed a website [14] where operators can validate links discovered by D-Miner. We received responses from three operators, for a total of 20 links. 18 validated correctly while two were declared as false. We re-conducted paris traceroute measurements to the destinations corresponding to the two false positive links. These links were found between the penultimate and the last IP seen in the traces. The last IP, which was not the destination IP, was repeating on all the subsequent TTLs until 30. Our interpretation is that this error was due to a routing loop or misconfiguration.

5 Forwarding path dynamics

With D-Miner, we reveal aspects of Internet dynamics that were under estimated [26] by the research community. 28.6% of D-Miner's probes saw their reply's IP address change, despite the flow identifier remaining constant over our three August 2019 snapshots. This would seem at first glance to be a startlingly high figure, implying more extensive routing changes than one might expect throughout the Internet [26,38]. But we provide evidence that, rather than routing changes, another phenomenon that we term *load balancer remapping* was responsible for at least 52% of these changes. These observations imply that future work should be cautious in attributing an observed `traceroute` change to a routing change.

5.1 Taxonomy of probe changes

A probe is uniquely identified by its (flow ID,TTL) pair. We say that there is a *probe change* if a probe elicits a reply from a different IP address in snapshot $i+1$ than in snapshot i. There may be as many probe changes as there are probes. We distinguish between meaningful and trivial changes.

To begin with, we do not count an *absence of response* as a meaningful change. That is, if a probe elicits a response in snapshot i and there is no reply to the probe in snapshot $i+1$, this difference may be attributable to loss or rate-limiting and is not indicative of a change. Similarly, we ignore absence in one round followed by a response in the next.

We are particularly interested in examining each probe change from the perspective of its predecessor node. Consider a probe (X,h) with flow ID X, sent with TTL h, revealing a vertex v_1; and a probe $(X,h+1)$ revealing a vertex v_2. If, in the next snapshot, (X,h) reveals v_1 again but $(X,h+1)$ reveals v_3, it is reasonable to infer that a mechanism at the router with interface v_1 is responsible for this probe change. Perhaps there has been a routing change, and the routing table at that router has been updated. Or, and this is the possibility that we focus on here: perhaps there has been no routing change, but instead that router is a load balancing router and the probe change results from a new load balancing decision for packets with flow ID X.

Previous research has identified *per-packet load balancing* [18] where a router simply disregards the flow ID X in its load balancing decision, for instance directing some packets to v_2, some to v_3, and some, perhaps, to other neighboring vertices, in a round-robin or (pseudo-)random fashion. As §3.5 describes, D-Miner tests for per-packet load balancing, and we exclude any reply variation due to that mechanism from our accounting of meaningful probe changes.

A possibility that previous literature has not explored is that a probe change could result from a per-flow or a per-destination load balancer making a load balancing change rather than a routing change. As an example, consider probe packets with flow IDs X and Y that both have the same destination IP address d. In snapshots i and $i+1$, probes (X,h) and (Y,h) both elicit replies from v_1, whereas in snapshot i, probe $(X,h+1)$ elicits a reply from v_2 and probe $(Y,h+1)$ from v_3, and in snapshot $i+1$, the replies are reversed. From one snapshot to the next, there has been no routing change for d at v_1: packets with that destination address continue to be load balanced across v_2 and v_3. But suppose the test for per-packet load balancing at v_1 has failed. We are left to consider that the probe change results from an update to the hashing decision that assigns flow IDs to next hop IP addresses. This is what we term *load balancer remapping*.

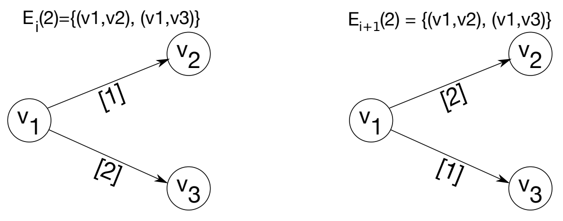
5.2 Load balancer remapping

Production systems are more nuanced in their behavior relative to the overview of load balancing in §2.1. For deterministic load balancing, in addition to header fields used to compute the hash function, a seed is generally added to avoid a phenomenon called *load balancing polarization* [1,11,13], which occurs when load balancers are chained and apply the same hashing algorithm, potentially causing load imbalance.

We experimented with Cisco routers running IOS 12 in a lab environment and confirmed that this seed is configurable. If the Cisco router reboots and has no saved seed, it generates a new random seed. For Juniper and Huawei, documentation tells us that they also use a seed [11,13].

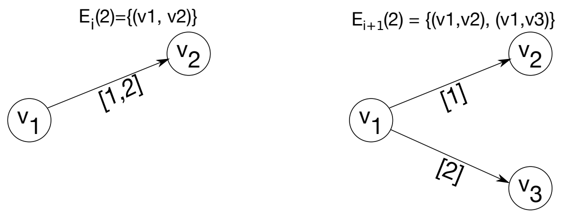
Thus, at least for Cisco, Juniper, and Huawei routers, a flow that took a certain load balanced path before a reboot can take a very different path after the reboot. Moreover, removing or adding an interface on a load-balancing router(s) toward a destination can cause the path assignment to be recalculated [8,10]. The particulars of the load balancing implementations in production can cause the reply IP address of a probe to change, *even when no routing change has occurred, and the flow identifier is constant*.

We attempt to identify in our dataset probe changes that correspond specifically to load balancer remapping. Our conservative rule of thumb is to say that if we observe a meaningful change for probe $(X,h+1)$, but at least one of the

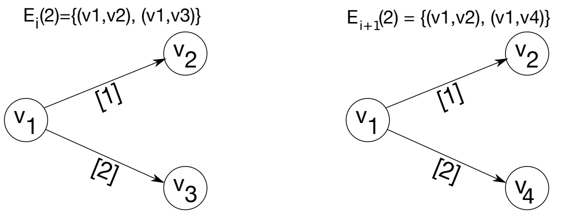


$E_i(2)=\{(v1,v2), (v1,v3)\}$
$E_{i+1}(2) = \{(v1,v2), (v1,v3)\}$
v_1
v_2
v_3
[1]
[2]
[2]
[1]

(a) Remapping, no new edges ($E_i = E_{i+1}$)



(b) Remapping, new edge ($E_i \subset E_{i+1}$)



$E_i(2)=\{(v1,v2), (v1,v3)\}$
$E_{i+1}(2) = \{(v1,v2), (v1,v4)\}$
v_1
v_2
v_3
v_4
[1]
[2]
[1]
[2]

(c) Remapping, new and deleted edges $(E_i \subsetneq E_{i+1}) \wedge (E_{i+1} \subsetneq E_i)$

Figure 7: Examples of three different types of changes for probe 2 across two snapshots i (left) and $i+1$ (right). In all cases, $E_i \cap E_{i+1} \neq \emptyset \implies$ we infer these as remapping events.

edges of the predecessor vertex is the same between snapshots, then remapping is taking place. If no edges are the same, it could still be the result of load balancer remapping instead of a routing change, but without evidence to allow us to infer this. Thus, our observations will underestimate the ubiquity of remapping.

We define $E_i(p)$ to be the set of out edges in snapshot i from the predecessor vertex of a meaningful change for probe $p = (X, h)$ (again, where a probe is a flow ID X and TTL h tuple). For each candidate meaningful-probe-change, we infer that the change is due to remapping if $E_i(p) \cap E_{i+1}(p) \neq \emptyset$.

Several classes of meaninful changes can be more more precisely characterized into subcategories. Either $E_i(p)$ and $E_{i+1}(p)$ can be: (1) equals, meaning that the changes might be due to a router reboot ($E_i = E_{i+1}$); (2) one set is included in the other, corresponding to potential addition/removal of some load balanced paths ($E_i \subset E_{i+1}$ or $E_{i+1} \subset E_i$); or (3) the sets have elements in common, but no inclusion relation can be established ($(E_i \subsetneq E_{i+1}) \wedge (E_{i+1} \subsetneq E_i)$), e.g., when load balanced paths are both added and removed.

Figure 7 illustrates these three cases by depicting of remapping from one snapshot, i (left), to the next, $i+1$ (right). Consider probe 2 in Fig 7a which is a meaningful change since it elicits v_3 in the first snapshot and v_2 in the second. The edges from the predecessor of these changes are the same between snapshots, i.e., $E_i(2) = E_{i+1}(2) = (v_1, v_2), (v_1, v_3)$. Note that the reciprocal change is observed by the flow 1 probe in $i+1$, for a total of two inferred remapping events.

In the example of Figure 7b, probe 2 is again a meaningful change, but in this instance elicits a new vertex v_3 in the second snapshot. In this case, the predecessor edges in the first snapshot are a subset of the second, i.e., $E_i(2) = (v_1, v_2)$ and $E_{i+1}(2) = (v_1, v_2), (v_1, v_3)$. Finally, Figure 7c shows an example of no inclusion relation where there is both a new and deleted edge due to remapping.

Table 3: Changes per (flow ID, TTL) probe on three snapshots.

Changes	0	1	2
Count	2,184,277,681	652,241,148	223,614,385
Total	3,060,133,214		

Table 4: Relation between the sets of edges where there have been probe changes.

$E_i \cap E_{i+1} = \emptyset$	$E_i \cap E_{i+1} \neq \emptyset$			
	$E_i = E_{i+1}$	$E_i \subsetneq E_{i+1}$	$E_i \supsetneq E_{i+1}$	Other
478,380,414	52,241,971	89,118,791	79,239,945	301,327,548
	521,928,255			
Total: 1,000,308,669				

5.3 Remapping observed

We now quantify probe changes and remapping observed across our three snapshots of §4.1. Because identifying remapping requires observing probing changes between two consecutive snapshots, we restrict our analysis to the set of probes with replies from two consecutive snapshots. Of the 6,019,578,262 unique flow IDs that elicited replies in our database (11,689,101,599 replies in total), we find that 3,060,133,214 of them are present in two consecutive snapshots, representing a bit more than half of the flow IDs.

To understand why nearly half of the probe replies do not appear in two consecutive snapshots, we find that for 87% (2,576,532,888) of them the probe ID has in fact been sent only in one snapshot. This is due to the adaptive nature of the D-Miner algorithm resulting in variations of discovery between snapshots presented in §4.4. If, for example, the number of edges discovered for a prefix differs across the three snapshots, the statistical guarantees tell us the number of probes that must be sent will also differ. This results in some probe IDs being sent in only one snapshot. The result is that we are likely underestimating the number of meaningful probe changes. For the remaining 13.0%, the probe was sent in two consecutive snapshots, but did not elicit two replies. This is likely due to normal packet losses in the network and possibly ICMP rate limiting [40].

We start by looking at the number of probe changes. Table 3 shows the distribution of the number of probe changes per probe. We see that 28.6% of the probes have at least one probe change. This number seems unusual if we were to consider it all as routing changes.Table 4 explains the previous 28.6% fraction by providing the distribution of 1,000,308,669 changes according to the remapping classification. The "Other" column refers to changes with no inclusion relationship but element(s) in common. Notice that the sum of these classifications is slightly smaller than total number of probe changes. The missing probe changes correspond to cases where there was no predecessor for the node corresponding to the IP reply elicited by the probe. This would happen in Figure 7 if v_1 was anonymous (a '*') for example.

We see that 52.2% of the probe changes correspond to remapping, which temper the impact of the 28.6% probe changes on routing changes. Finally, for each of the probe changes corresponding to remapping, we performed IP to AS translation on the corresponding IP reply, using August 4, 2019 BGP data from route views [5]. We found that remapping events were spread over 39,455 ASes, showing that this phenomenon is widespread. We conclude this section by noting that all of the changes between snapshots, either due to D-Miner varying the set of probes from one snapshot to another; real dynamics such as routing changes; or remapping due to reboots and/or adding/removing load balanced paths, make any efficiency optimization based on historical discovery hard. It also further corroborates Cunha et al.'s finding that it is difficult to predict path changes [27].

6 An updated Internet load balancing survey

Its been eight years since the last published survey of load balancing in the Internet [18]. Whereas this previous study was limited to MDA traceroutes to ~120k targets, in this section we undertake the task of leveraging D-Miner to perform an exhaustive survey of Internet load balancing on 14,461,947 /24 destination prefixes.

Some things have certainly changed. We now see far larger load balanced topologies, for instance, with thousands of edges, instead of tens. This section updates our understanding of load balancing in the Internet, with some of the more notable results being: 17.9% of load balancing takes place between autonomous systems (i.e., *inter-AS load balancing*); just one autonomous system accounts for the topologies that contain more than 2000 edges; 64.7% of our traces towards all of the /24 prefixes contain at least one branching point (but this might be vantage point dependent); and 1.9% of branching points are per-packet load balancers.

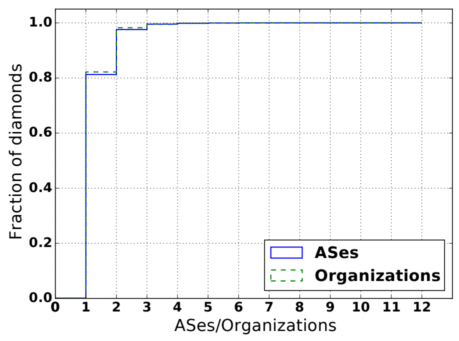


Figure 8: Intra- and inter-AS/organization load balancing.

6.1 Dataset

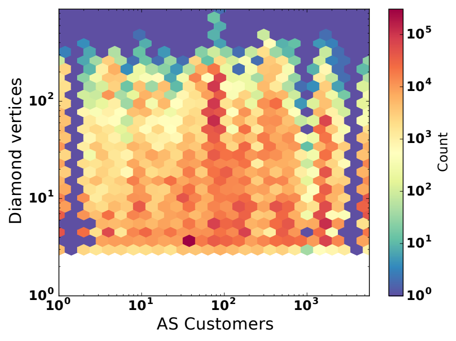
From the first snapshot of §4.1 (Aug 6-8, 2019), we extract all of the unique "diamonds" found on the load balanced paths. We adopt the same definition of a diamond as given by Augustin et al. [18]: a *diamond* is "a subgraph delimited by a divergence point followed, two or more hops later, by a convergence point, with the requirement that all flows from source to destination flow through both points." We say that two diamonds are equal if they share the same divergence and convergence points. When the divergence point or the convergence point is a $\star$ (i.e., this TTL is "anonymous"), we say that two diamonds are equal if they have identical node sets. In sending probes towards all IPv4 /24 destination prefixes, we extracted 4,029,866 unique diamonds.

6.2 Intra- and inter-AS load balancing

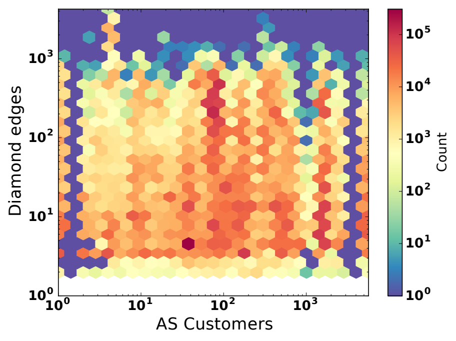
Augustin et al. found just one instance of inter-AS load balancing in their 2011 survey [18], whereas we now find it to be a more prevalent practice. We use Oregon Route Views BGP data [5] from 4 August 2019 to map IP addresses of the router interfaces comprising diamonds we discover in the Internet to autonomous systems (ASes). We further map AS numbers to organization names using CAIDA's AS Rank [33].

IP address to AS mapping is an outstanding research problem, and correct attribution is known to be difficult [37]. For instance, the IP address of a customer or peer border router is frequently allocated from her provider or peer's address space. We therefore adopt the same methodology of Augustin et al. [18] of not including the diamond's divergence or convergence point when determining the diamond's AS composition. Thus, our estimates of inter-AS load balancing are intended to be conservative.

The CDFs of Figure 8 show that, while most load balancing still takes place within a single autonomous system (AS) or organization, a significant portion takes place across two or more of them: 18.7% for ASes and 17.9% for organizations. In one case, we found a single diamond with addresses from 12 ASes (explaining why the CDF continues to 12).



(a) Vertices



(b) Edges

Figure 9: Joint distribution between the number of customers of ASes and size of the diamonds in these ASes.

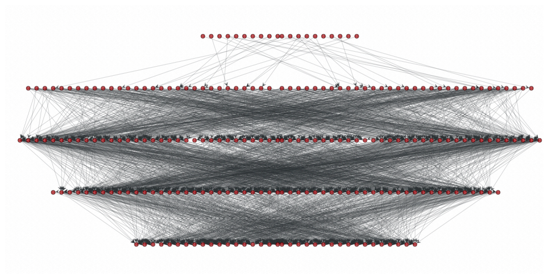


Figure 10: Extract of D-Miner trace to an Amazon /24 prefix.

6.3 ASes that host load balancers

We next investigate the relationship between the size of an AS (measured as number of customers) and the prevalence of load balancing in that AS. We use AS Rank from CAIDA [33] to perform the AS to customers mapping. When there is more than one AS in the diamond, each AS in the diamond is counted one time. Figure 9 shows no clear correlation between the number of customers in an AS and the amount of load balancing we infer, suggesting that load balancing is not limited to large networks, but is a widespread phenomenon.

However, there are some extreme cases involving large networks. We find 74 diamonds of more than 500 nodes each in the network of a French mobile network operator (SFR). Other ASes contain diamonds with $>10^3$ diamond edges, as seen in Figure 9b. There are 8,407 diamonds with more than 2,000 edges, of which 8,400 belong to Amazon – likely entry points to their datacenters. The DNS PTR records for the diamond's IP addresses are variations of the address and location, e.g., `ec2-54-178-57-0.ap-northeast-1.compute.amazonaws.com`. These names are characteristic of Amazon's cloud infrastructure. An example of such of a trace is shown in Figure 10. This figure shows the last TTLs of D-Miner tracing from a single VP to a single /24 prefix belonging to Amazon. The topology itself strongly resembles current data center design practices that use Clos architectures [47]; we requested validation from Amazon, however, they were unable to provide any corroborating information due to their internal privacy polices. This example, one of thousands in our data, shows just how complex load balanced topologies can be – complexity that would otherwise be missed without the comprehensive multipath mapping D-Miner provides.

7 Conclusion and future work

In this work we present D-Miner, the first Internet-scale system that captures a multipath view of the topology. By combining and adapting state-of-the-art multipath detection and high speed randomized topology discovery techniques, D-Miner permits discovery of the Internet's multipath topology in 2.5 days when probing at 100kpps. This high speed allows us to characterize and quantify dynamic behaviors of the Internet induced by load balancing. Finally, D-Miner enables for the first time an Internet Scale survey of load balancing that shows its widespread prevalence, both in the core and at the edge. We release the D-Miner source code and make our datasets publicly available.

Our hope is that D-Miner and our data will facilitate better understanding of the Internet's true structure, properties, and resilience. Future work includes running D-Miner with other transport protocols to compare load balancing usage between them at Internet scale, as well as adapting D-Miner to IPv6.

Our empirical data suggests that there are a set of load balanced architectures common to different provider types, for instance those deployed in data centers versus mobile operators versus transit providers. We believe there is significant opportunity to develop a taxonomy of these common architectures and classify results accordingly, as well as to identify previously unidentified load balanced architectures that are deployed in the wild. Comprehensive mapping of some of these topologies may require probing both from outside and within the provider's network; we leave an exploration of e.g., internal data center probing to future work.

Finally, in order to provide regular surveys to the community, we wish to deploy D-Miner on more vantage points at high probing speed, create periodic snapshots and perform alias resolution on the resulting discovered topologies.

Acknowledgments

We thank: Ophelie Walrick, for suggesting the name Diamond-Miner; Matthieu Gouel, for the website for topology validation; network administrators at Sorbonne Université and the Renater NREN, for assistance in running our measurements; Romain Fontugne, from IIJ, Niels den Otter, from SURFNet, Bernd Spiess, from IP-IT, Olva Kvittem, from UNINETT, and Marc Ammann, from SUNRISE, for their ground truth validation; the anonymous reviewers from NSDI, and our shepherd, Ben Zhao, for their careful reading and suggestions. Robert Beverly and Justin P. Rohrer are associated with Naval Postgraduate School, department of Computer Science. Kevin Vermeulen, Olivier Fourmaux, and Timur Friedman are associated with Sorbonne Université, CNRS, Laboratoire d'informatique de Paris 6, LIP6, F-75005 Paris, France. Kevin Vermeulen and Timur Friedman are associated with the Laboratory of Information, Networking and Communication Sciences, LINCS, F-75013 Paris, France. Vermeulen, Rohrer, and Beverly were supported in part by the Laboratory for Telecommunication Sciences. Vermeulen, Fourmaux, and Friedman were supported in part by a university research grant from the French Ministry of Defense.

References

[1] CEF polarization. https://www.cisco.com/c/en/us/support/docs/ip/express-forwarding-cef/116376-technote-cef-00.html.

[2] Clickhouse — open source distributed column-oriented DBMS. https://clickhouse.yandex.

[3] Configuring a load-balancing scheme. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipswitch_cef/configuration/xe-3s/isw-cef-xe-3s-book/isw-cef-load-balancing.pdf.

[4] D-miner topology snapshots. https://gitlab.planet-lab.eu/cartography/.

[5] Oregon route views. http://www.routeviews.org/.

[6] Per-flow and per-packet load balancing. https://support.huawei.com/enterprise/en/doc/EDOC1100055041/ebc8ad42.

[7] PlanetLab Europe. https://www.planet-lab.eu.

[8] Resilient hashing on LAGs and ECMP groups. https://www.juniper.net/documentation/en_US/junos/topics/topic-map/switches-interface-resilient-hashing.html.

[9] Traceroute for linux. http://traceroute.sourceforge.net.

[10] Troubleshooting load balancing over parallel links using cisco express forwarding. https://www.cisco.com/c/en/us/support/docs/ip/express-forwarding-cef/18285-loadbal-cef.html#internalmech.

[11] Understanding the algorithm used to hash LAG bundle and egress next-hop ECMP traffic (QFX 10002 and QFX 10008 switches). https://www.juniper.net/documentation/en_US/junos/topics/concept/interfaces-hashing-lag-ecmp-understanding-\10002-10008.html.

[12] Understanding the algorithm used to load balance traffic on MX series routers. https://www.juniper.net/documentation/en_US/junos/topics/concept/hash-computation-mpcs-understanding.html.

[13] Understanding the hash algorithm. https://support.huawei.com/enterprise/en/doc/EDOC1100086965.

[14] Website for topology validation . http://heartbeat.planet-lab.eu:8000.

[15] Bernhard Ager, Nikolaos Chatzis, Anja Feldmann, Nadi Sarrar, Steve Uhlig, and Walter Willinger. Anatomy of a large European IXP. In *Proc. ACM SIGCOMM '12*. ACM, 2012.

[16] Brice Augustin, Timur Friedman, and Renata Teixeira. Exhaustive path tracing with Paris traceroute. In *Proc. ACM CoNEXT '07*, 2006.

[17] Brice Augustin, Timur Friedman, and Renata Teixeira. Multipath tracing with Paris traceroute. In *Proc. E2EMON '07*, 2007.

[18] Brice Augustin, Timur Friedman, and Renata Teixeira. Measuring multipath routing in the Internet. *IEEE/ACM Transactions on Networking*, 19(3):830–840, June 2011.

[19] Robert Beverly. Yarrp'ing the Internet: Randomized high-speed active topology discovery. In *Proc. ACM IMC '16*, 2016.

[20] Robert Beverly, Ramakrishnan Durairajan, David Plonka, and Justin P Rohrer. In the IP of the beholder: Strategies for active IPv6 topology discovery. In *Proc. ACM IMC '18*, 2018.

[21] CAIDA. The CAIDA ark IPv4 Internet topology data kits dataset, 2019. http://www.impactcybertrust.org.

[22] Kenneth L Calvert, Matthew B Doar, and Ellen W Zegura. Modeling Internet topology. *IEEE Communications magazine*, 35(6):160–163, 1997.

[23] Zhiruo Cao, Zheng Wang, and Ellen Zegura. Performance of hashing-based schemes for Internet load balancing. In *Proc. INFOCOM '00*, pages 332–341, 2000.

[24] kc claffy, Young Hyun, Ken Keys, Marina Fomenkov, and Dmitri Krioukov. Internet mapping: From art to science. In *IEEE DHS Cybersecurity Applications and Technologies Conference for Homeland Security (CATCH)*, 2009.

[25] M. Cotton, L. Vegoda, R. Bonica, and B. Haberman. Special-purpose IP address registries. RFC 6890 (Best Current Practice), April 2013.

[26] Ítalo Cunha, Renata Teixeira, and Christophe Diot. Measuring and Characterizing End-to-End Route Dynamics in the Presence of Load Balancing. In *Proc. PAM '11*, 2011.

[27] Ítalo Cunha, Renata Teixeira, Darryl Veitch, and Christophe Diot. DTRACK: A system to predict and track Internet path changes. *IEEE/ACM Transactions on Networking*, 22(4):1025–1038, August 2014.

[28] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the Internet topology. *ACM SIGCOMM Computer Communication Review*, 29(4):251–262, 1999.

[29] Christian E Hopps. Analysis of an equal-cost multi-path algorithm. RFC 2992 (Informational), November 2000.

[30] Van Jacobson. 4BSD routing diagnostic tool available for ftp. Email 8812201313.AA03127@helios.ee.lbl.gov to the IETF and end2end-interest e-mail lists, 1988.

[31] Ethan Katz-Bassett, John P. John, Arvind Krishnamurthy, David Wetherall, Thomas Anderson, and Yatin Chawathe. Towards IP geolocation using delay and topology measurements. In *Proc. ACM IMC '06*, 2006.

[32] Rupa Krishnan, Harsha V. Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize CDN performance. In *Proc. ACM IMC '09*, 2009.

[33] M. Luckie, B. Huffaker, k. claffy, A. Dhamdhere, and V. Giotsas. AS relationships, customer cones, and validation. In *Proc. ACM IMC '13*, 2013.

[34] Matthew Luckie. Scamper: A scalable and extensible packet prober for active measurement of the Internet. In *Proc. ACM IMC '10*, 2010.

[35] Matthew Luckie and Robert Beverly. The impact of router outages on the AS-level Internet. In *Proc. ACM SIGCOMM '17*, 2017.

[36] Matthew Luckie, Young Hyun, and Bradley Huffaker. Traceroute probe method and forward IP path inference. In *Proc. ACM IMC '08*, 2008.

[37] Alexander Marder, Matthew Luckie, Amogh Dhamdhere, Bradley Huffaker, Jonathan M Smith, et al. Pushing the boundaries with bdrmapIT: Mapping router ownership at Internet scale. In *Proc. ACM IMC '18*, 2018.

[38] Vern Paxson. End-to-end routing behavior in the internet. *IEEE/ACM transactions on Networking*, 5(5):601–615, 1997.

[39] Vern Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[40] Riccardo Ravaioli, Guillaume Urvoy-Keller, and Chadi Barakat. Characterizing ICMP rate limitation on routers. In *Proc. ICC '15*, 2015.

[41] RIPE NCC. RIPE Atlas, 2019. `https://atlas.ripe.net/`.

[42] Stephen D. Strowes. Visibility of IPv4 and IPv6 Prefix Lengths in 2019, 2019. `https://labs.ripe.net/Members/stephen_strowes/visibility-of-prefix-lengths-in-ipv4-and-ipv6`.

[43] James P.G. Sterbenz, Egemen K. Çetinkaya, Mahmood A. Hameed, Abdul Jabbar, Qian Shi, and Justin P. Rohrer. Evaluation of network resilience, survivability, and disruption tolerance: Analysis, topology generation, simulation, and experimentation. *Springer Telecommunication Systems*, 52(2):705–736, February 2011.

[44] Darryl Veitch, Brice Augustin, Renata Teixeira, and Timur Friedman. Failure control in multipath route tracing. In *Proc. IEEE Infocom '09*, 2009.

[45] Kevin Vermeulen, Stephen D Strowes, Olivier Fourmaux, and Timur Friedman. Multilevel MDA-Lite Paris traceroute. In *Proc. ACM IMC '18*. ACM, 2018.

[46] Walter Willinger, David Alderson, and John C Doyle. Mathematics and the Internet: A source of enormous confusion and great potential. *Notices of the American Mathematical Society*, 56(5):586–599, 2009.

[47] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding Lifecycle Management Complexity of Datacenter Topologies. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019.

[48] Zheng Zhang, Ying Zhang, Y. Charlie Hu, Z. Morley Mao, and Randy Bush. iSPY: Detecting IP prefix hijacking on my own. In *Proceedings of the ACM SIGCOMM*

2008 Conference on Data Communication, SIGCOMM '08, pages 327–338, New York, NY, USA, 2008. ACM.

Learning *in situ*: a randomized experiment in video streaming

Francis Y. Yan, Hudson Ayers, Chenzhi Zhu[†], Sadjad Fouladi
James Hong, Keyi Zhang, Philip Levis, Keith Winstein

Stanford University, [†]Tsinghua University

Abstract

We describe the results of a randomized controlled trial of video-streaming algorithms for bitrate selection and network prediction. Over the last year, we have streamed 38.6 years of video to 63,508 users across the Internet. Sessions are randomized in blinded fashion among algorithms.

We found that in this real-world setting, it is difficult for sophisticated or machine-learned control schemes to outperform a "simple" scheme (buffer-based control), notwithstanding good performance in network emulators or simulators. We performed a statistical analysis and found that the heavy-tailed nature of network and user behavior, as well as the challenges of emulating diverse Internet paths during training, present obstacles for learned algorithms in this setting.

We then developed an ABR algorithm that robustly outperformed other schemes, by leveraging data from its deployment and limiting the scope of machine learning only to making predictions that can be checked soon after. The system uses supervised learning *in situ*, with data from the real deployment environment, to train a probabilistic predictor of upcoming chunk transmission times. This module then informs a classical control policy (model predictive control).

To support further investigation, we are publishing an archive of data and results each week, and will open our ongoing study to the community. We welcome other researchers to use this platform to develop and validate new algorithms for bitrate selection, network prediction, and congestion control.

1 Introduction

Video streaming is the predominant Internet application, making up almost three quarters of all traffic [41]. One key algorithmic question in video streaming is *adaptive bitrate selection*, or ABR, which decides the compression level selected for each "chunk," or segment, of the video. ABR algorithms optimize the user's quality of experience (QoE): more-compressed chunks reduce quality, but larger chunks may stall playback if the client cannot download them in time.

In the academic literature, many recent ABR algorithms use statistical and machine-learning methods [4, 25, 38–40, 46], which allow algorithms to consider many input signals and try to perform well for a wide variety of clients. An ABR decision can depend on recent throughput, client-side buffer occupancy, delay, the experience of clients on similar ISPs or types of connectivity, etc. Machine learning can find patterns in seas of data and is a natural fit for this problem domain.

However, it is a perennial lesson that the performance of learned algorithms depends on the data or environments used to train them. ML approaches to video streaming and other wide-area networking challenges are often hampered in their access to good and representative training data. The Internet is complex and diverse, individual nodes only observe a noisy sliver of the system dynamics, and behavior is often heavy-tailed and changes with time. Even with representative throughput traces, accurately simulating or emulating the diversity of Internet paths requires more than replaying such traces and is beyond current capabilities [15, 16, 31, 45].

As a result, the performance of algorithms in emulated environments may not generalize to the Internet [7]. For example, CS2P's gains were more modest over real networks than in simulation [40]. Measurements of Pensieve [25] saw narrower benefits on similar paths [11] and a large-scale streaming service [24]. Other learned algorithms, such as the Remy congestion-control schemes, have also seen inconsistent results on real networks, despite good results in simulation [45].

This paper seeks to answer: *what does it take to create a learned ABR algorithm that robustly performs well over the wild Internet?* We report the design and findings of Puffer[1], an ongoing research study that operates a video-streaming website open to the public. Over the past year, Puffer has streamed 38.6 years of video to 63,508 distinct users, while recording client telemetry for analysis (current load is about 60 stream-days of data per day). Puffer randomly assigns each session to one of a set of ABR algorithms; users are blinded to the assignment. We find:

[1] https://puffer.stanford.edu

In our real-world setting, sophisticated algorithms based on control theory [46] or reinforcement learning [25] did not outperform simple buffer-based control [18]. We found that more-sophisticated algorithms do not necessarily beat a simpler, older algorithm. The newer algorithms were developed and evaluated using throughput traces that may not have captured enough of the Internet's heavy tails and other dynamics when replayed in simulation or emulation. Training them on more-representative traces doesn't necessarily reverse this: we retrained one algorithm using throughput traces drawn from Puffer (instead of its original set of traces) and evaluated it also on Puffer, but the results were similar (§5.3).

Statistical margins of error in quantifying algorithm performance are considerable. Prior work on ABR algorithms has claimed benefits of 10–15% [46], 3.2–14% [40], or 12–25% [25], based on throughput traces or real-world experiments lasting hours or days. However, we found that the empirical variability and heavy tails of throughput evolution and rebuffering create statistical margins of uncertainty that make it challenging to detect real effects of this magnitude. Even with a *year* of experience per scheme, a 20% improvement in rebuffering ratio would be statistically indistinguishable, i.e., below the threshold of detection with 95% confidence. These uncertainties affect the design space of machine-learning approaches that can practically be deployed [13, 26].

It is possible to robustly outperform existing schemes by combining classical control with an ML predictor trained *in situ* on real data. We describe Fugu, a data-driven ABR algorithm that combines several techniques. Fugu is based on MPC (model predictive control) [46], a classical control policy, but replaces its throughput predictor with a deep neural network trained using supervised learning on data recorded *in situ* (in place), meaning from Fugu's actual deployment environment, Puffer. The predictor has some uncommon features: it predicts *transmission time* given a chunk's file size (vs. estimating throughput), it outputs a probability distribution (vs. a point estimate), and it considers low-level congestion-control statistics among its input signals. Ablation studies (§4.2) find each of these features to be necessary to Fugu's performance.

In a controlled experiment during most of 2019, Fugu outperformed existing techniques—including the simple algorithm—in stall ratio (with one exception), video quality, and the variability of video quality (Figure 1). The improvements were significant both statistically and, perhaps, practically: users who were randomly assigned to Fugu (in blinded fashion) chose to continue streaming for 5–9% longer, on average, than users assigned to the other ABR algorithms.[2]

Our results suggest that, as in other domains, good and representative training is the key challenge for robust performance of learned networking algorithms, a somewhat different point of view from the generalizability arguments in prior

[2]This effect was driven solely by users streaming more than 3 hours of video; we do not fully understand it.

Results of primary experiment (Jan. 26–Aug. 7 & Aug. 30–Oct. 16, 2019)

Algorithm	Time stalled (lower is better)	Mean SSIM (higher is better)	SSIM variation (lower is better)	Mean duration (time on site)
Fugu	0.13%	16.64 dB	0.74 dB	33.6 min
MPC-HM [46]	0.22%	16.61 dB	0.79 dB	30.8 min
BBA [18]	0.19%	16.56 dB	1.11 dB	32.1 min
Pensieve [25]	0.17%	16.26 dB	1.05 dB	31.6 min
RobustMPC-HM	0.12%	16.01 dB	0.98 dB	31.0 min

Figure 1: In an eight-month randomized controlled trial with blinded assignment, the Fugu scheme outperformed other ABR algorithms. The primary analysis includes 637,189 streams played by 54,612 client IP addresses (13.1 client-years in total). Uncertainties are shown in Figures 9 and 11.

work [25, 37, 44]. One way to achieve representative training is to learn in place (*in situ*) on the actual deployment environment, assuming the scheme can be feasibly trained this way and the deployment is widely enough used to exercise a broad range of scenarios.[3] The approach we describe here is only a step in this direction, but we believe Puffer's results suggest that learned systems will benefit by addressing the challenge of "*how will we get enough representative scenarios for training—what is enough, and how do we keep them representative over time?*" as a first-class consideration.

We intend to operate Puffer as an "open research" project as long as feasible. We invite the research community to train and test new algorithms on randomized subsets of its traffic, gaining feedback on real-world performance with quantified uncertainty. Along with this paper, we are publishing an archive of data and results back to the beginning of 2019 on the Puffer website, with new data and results posted weekly.

In the next few sections, we discuss the background and related work on this problem (§2), the design of our blinded randomized experiment (§3) and the Fugu algorithm (§4), with experimental results in Section 5, and a discussion of results and limitations in Section 6. In the appendices, we provide a standardized diagram of the experimental flow for the primary analysis and describe the data we are releasing.

2 Background and related work

The basic problem of adaptive video streaming has been the subject of much academic work; for a good overview, we refer the reader to Yin et al. [46]. We briefly outline the problem here. A service wishes to serve a pre-recorded or live video stream to a broad array of clients over the Internet. Each client's connection has a different and unpredictable time-varying performance. Because there are many clients, it is not feasible for the service to adjust the encoder configuration in real time to accommodate any one client.

[3]Even collecting traces from a deployment environment and replaying them in a simulator or emulator to train a control policy—as is typically necessary in reinforcement learning—is not what we mean by "*in situ*."

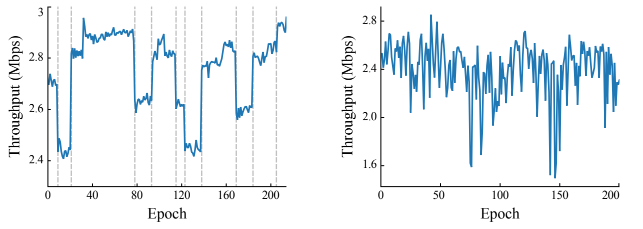
Instead, the service encodes the video into a handful of alternative compressed versions. Each represents the original video but at a different quality, target bitrate, or resolution. Client sessions choose from this limited menu. The service encodes the different versions in a way that allows clients to switch midstream as necessary: it divides the video into *chunks*, typically 2–6 seconds each, and encodes each version of each chunk independently, so it can be decoded without access to any other chunks. This gives clients the opportunity to switch between different versions at each chunk boundary. The different alternatives are generally referred to as different "bitrates," although streaming services today generally use "variable bitrate" (VBR) encoding [32], where within each alternative stream, the chunks vary in compressed size [47].

Choosing which chunks to fetch. Algorithms that select which alternative version of each chunk to fetch and play, given uncertain future throughput, are known as *adaptive bitrate* (ABR) schemes. These schemes fetch chunks, accumulating them in a playback buffer, while playing the video at the same time. The playhead advances and drains the buffer at a steady rate, 1 s/s, but chunks arrive at irregular intervals dictated by the varying network throughput and the compressed size of each chunk. If the buffer underflows, playback must stall while the client "rebuffers": fetching more chunks before resuming playback. The goal of an ABR algorithm is typically framed as choosing the optimal sequence of chunks to fetch or replace [38], given recent experience and guesses about the future, to minimize startup time and presence of stalls, maximize the quality of chunks played back, and minimize variation in quality over time (especially abrupt changes in quality). The importance tradeoff for these factors is captured in a QoE metric; several studies have calibrated QoE metrics against human behavior or opinion [6, 12, 21].

Adaptive bitrate selection. Researchers have produced a literature of ABR schemes, including "rate-based" approaches that focus on matching the video bitrate to the network throughput [20, 23, 27], "buffer-based" algorithms that steer the duration of the playback buffer [18, 38, 39], and control-theoretic schemes that try to maximize expected QoE over a receding horizon, given the upcoming chunk sizes and a prediction of the future throughput.

Model Predictive Control (MPC), FastMPC, and RobustMPC [46] fall into the last category. They comprise two modules: a *throughput predictor* that informs a predictive *model* of what will happen to the buffer occupancy and QoE in the near future, depending on which chunks it fetches, with what quality and sizes. MPC uses the model to plan a sequence of chunks over a limited horizon (e.g., the next 5–8 chunks) to maximize the expected QoE. We implemented MPC and RobustMPC for Puffer, using the same predictor as the paper: the harmonic mean of the last five throughput samples.

CS2P [40] and Oboe-tuned RobustMPC [4] are related to MPC; they constitute better throughput predictors that inform



(a) CS2P example session (Figure 4a from [40])

(b) Typical Puffer session with similar mean throughput

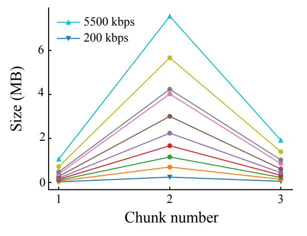
Figure 2: Puffer has not observed CS2P's discrete throughput states. (Epochs are 6 seconds in both plots.)

the same control strategy (MPC). These throughput predictors were trained on real datasets that recorded the evolution of throughput over time within a session. CS2P clusters users by similarity and models their evolving throughput as a Markovian process with a small number of discrete states; Oboe uses a similar model to detect when the network path has changed state. In our dataset, we have not observed CS2P and Oboe's observation of discrete throughput states (Figure 2).

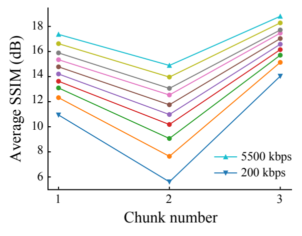
Fugu fits in this same category of algorithms. It also uses MPC as the control strategy, informed by a network predictor trained on real data. This component, which we call the Transmission Time Predictor (TTP), incorporates a number of uncommon features, none of which can claim novelty on its own. The TTP explicitly predicts the transmission time of a chunk with given size and isn't a "throughput" predictor *per se*. A throughput predictor models the transmission time of a chunk as scaling linearly with size, but it is well known that observed throughput varies with file size [7, 32, 47], in part because of the effects of congestion control and because chunks of different sizes experience different time intervals of the path's varying capacity. To our knowledge, Fugu is the first to use this fact operationally as part of a control policy.

Fugu's predictor is also *probabilistic*: it outputs not a single predicted transmission time, but a probability distribution on possible outcomes. The use of uncertainty in model predictive control has a long history [36], but to our knowledge Fugu is the first to use stochastic MPC in this context. Finally, Fugu's predictor is a neural network, which lets it consider an array of diverse signals that relate to transmission time, including raw congestion-control statistics from the sender-side TCP implementation [17, 42]. We found that several of these signals (RTT, CWND, etc.) benefit ABR decisions (§5).

Pensieve [25] is an ABR scheme also based on a deep neural network. Unlike Fugu, Pensieve uses the neural network not simply to make predictions but to make *decisions* about which chunks to send. This affects the type of learning used to train the algorithm. While CS2P and Fugu's TTP can be trained with *supervised learning* (to predict chunk transmission times recorded from past data), it takes more than data to train a scheme that makes decisions; one needs training *envi-*



(a) VBR encoding lets chunk size vary within a stream [47].



(b) Picture quality also varies with VBR encoding [32].

Figure 3: Variations in picture quality and chunk size within each stream suggest a benefit from choosing chunks based on SSIM and size, rather than average bitrate (legend).

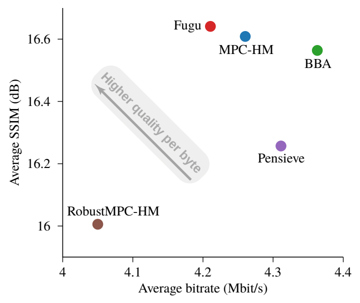


Figure 4: On Puffer, schemes that maximize average SSIM (MPC-HM, RobustMPC-HM, and Fugu) delivered higher quality video per byte sent, vs. those that maximize bitrate directly (Pensieve) or the SSIM of each chunk (BBA).

ronments that respond to a series of decisions and judge their consequences. This is known as reinforcement learning (RL). Generally speaking, RL techniques expect a set of training environments that can exercise a control policy through a range of situations and actions [3], and need to be able to observe a detectable difference in performance by slightly varying a control action. Systems that are challenging to simulate or that have too much noise present difficulties [13, 26].

3 Puffer: an ongoing live study of ABR

To understand the challenges of video streaming and measure the behavior of ABR schemes, we built Puffer, a free, publicly accessible website that live-streams six over-the-air commercial television channels. Puffer operates as a randomized controlled trial; sessions are randomly assigned to one of a set of ABR or congestion-control schemes. The study participants include any member of the public who wishes to participate. Users are blinded to algorithm assignment, and we record client telemetry on video quality and playback. A Stanford Institutional Review Board determined that Puffer does not constitute human subjects research.

Our reasoning for streaming live television was to collect data from enough participants and network paths to draw robust conclusions about the performance of algorithms for ABR control and network prediction. Live television is an evergreen source of popular content that had not been broadly available for free on the Internet. Our study benefits, in part, from a law that allows nonprofit organizations to retransmit over-the-air television signals without charge [1]. Here, we describe details of the system, experiment, and analysis.

3.1 Back-end: decoding, encoding, SSIM

Puffer receives six television channels using a VHF/UHF antenna and an ATSC demodulator, which outputs MPEG-2 transport streams in UDP. We wrote software to decode a stream to chunks of raw decoded video and audio, maintaining synchronization (by inserting black fields or silence) in the event of lost transport-stream packets on either substream. Video chunks are 2.002 seconds long, reflecting the 1/1001 factor for NTSC frame rates. Audio chunks are 4.8 seconds long. Video is de-interlaced with `ffmpeg` to produce a “canonical” 1080p60 or 720p60 source for compression.

Puffer encodes each video chunk in ten different H.264 versions, using `libx264` in `veryfast` mode. The encodings range from 240p60 video with constant rate factor (CRF) of 26 (about 200 kbps) to 1080p60 video with CRF of 20 (about 5,500 kbps). Audio chunks are encoded in the Opus format.

Puffer then uses `ffmpeg` to calculate each encoded chunk’s SSIM [43], a measure of video quality, relative to the canonical source. This information is used by the objective function of BBA, MPC, RobustMPC, and Fugu, and for our evaluation. In practice, the relationship between bitrate and quality varies chunk-by-chunk (Figure 3), and users cannot perceive compressed chunk sizes directly—only what is shown on the screen. ABR schemes that maximize bitrate do not necessarily see a commensurate benefit in picture quality (Figure 4).

Encoding six channels in ten versions each (60 streams total) with `libx264` consumes about 48 cores of an Intel x86-64 2.7 GHz CPU in steady state. Calculating the SSIM of each encoded chunk consumes an additional 18 cores.

3.2 Serving chunks to the browser

To make it feasible to deploy and test arbitrary ABR schemes, Puffer uses a “dumb” player (using the HTML5 `<video>` tag and the JavaScript Media Source Extensions) on the client side, and places the ABR scheme at the server. We have a 48-core server with 10 Gbps Ethernet in a datacenter at Stanford. The browser opens a WebSocket (TLS/TCP) connection to a daemon on the server. Each daemon is configured with a different TCP congestion control (for the primary analysis, we used BBR [9]) and ABR scheme. Some schemes are more efficiently implemented than others; on average the CPU load from serving client traffic (including TLS, TCP, and ABR) is about 5% of an Intel x86-64 2.7 GHz core per stream.

Algorithm	Control	Predictor	Optimization goal	How trained
BBA	classical (linear control)	*n/a*	$+$SSIM *s.t.* bitrate $<$ limit	*n/a*
MPC-HM	classical (MPC)	classical (HM)	$+\overline{\text{SSIM}}$, $-$stalls, $-\Delta$SSIM	*n/a*
RobustMPC-HM	classical (robust MPC)	classical (HM)	$+\overline{\text{SSIM}}$, $-$stalls, $-\Delta$SSIM	*n/a*
Pensieve	learned (DNN)	*n/a*	$+\overline{\text{bitrate}}$, $-$stalls, $-\Delta$bitrate	reinforcement learning in simulation
Fugu	classical (MPC)	learned (DNN)	$+\overline{\text{SSIM}}$, $-$stalls, $-\Delta$SSIM	supervised learning *in situ*

Figure 5: Distinguishing features of algorithms used in the primary experiment. HM = harmonic mean of last five throughput samples. MPC = model predictive control. DNN = deep neural network.

Sessions are randomly assigned to serving daemons. Users can switch channels without breaking their TCP connection and may have many "streams" within each session.

Puffer is not a client-side DASH [28] (Dynamic Adaptive Streaming over HTTP) system. Like DASH, though, Puffer is an ABR system streaming chunked video over a TCP connection, and runs the same ABR algorithms that DASH systems can run. We don't expect this architecture to replace client-side ABR (which can be served by CDN edge nodes), but we expect its conclusions to translate to ABR schemes broadly. The Puffer website works in the Chrome, Firefox, Edge, and Opera browsers, including on Android phones, but does not play in the Safari browser or on iOS (which lack support for the Media Source Extensions or Opus audio).

3.3 Hosting arbitrary ABR schemes

We implemented buffer-based control (BBA), MPC, RobustMPC, and Fugu in back-end daemons that serve video chunks over the WebSocket. We use SSIM in the objective functions for each of these schemes. For BBA, we use the formula in the original paper [18] to decide the maximum chunk size, and subject to this constraint, the chunk with the highest SSIM is selected to stream. We also choose reservoir values consistent with our 15-second maximum buffer.

Deploying Pensieve for live streaming. We use the released Pensieve code (written in Python with TensorFlow) directly. When a client is assigned to Pensieve, Puffer spawns a Python subprocess running Pensieve's multi-video model.

We contacted the Pensieve authors to request advice on deploying the algorithm in a live, multi-video, real-world setting. The authors recommended that we use a longer-running training and that we tune the entropy parameter when training the multi-video neural network. We wrote an automated tool to train 6 different models with various entropy reduction schemes. We tested these manually over a few real networks, then selected the model with the best performance. We modified the Pensieve code (and confirmed with the authors) so that it does not expect the video to end before a user's session completes. We were not able to modify Pensieve to optimize SSIM; it considers the average bitrate of each Puffer stream. We adjusted the video chunk length to 2.002 seconds and the buffer threshold to 15 seconds to reflect our parameters. For training data, we used the authors' provided script to generate 1000 simulated videos as training videos, and a combination of the FCC and Norway traces linked to in the Pensieve codebase as training traces.

3.4 The Puffer experiment

To recruit participants, we purchased Google and Reddit ads for keywords such as "live tv" and "tv streaming" and paid people on Amazon Mechanical Turk to use Puffer. We were also featured in press articles. Popular programs (e.g. the 2019 and 2020 Super Bowls, the Oscars, World Cup, and "Bachelor in Paradise") brought large spikes ($> 20\times$) over baseline load. Our current average load is about 60 concurrent streams.

Between Jan. 26, 2019 and Feb. 2, 2020, we have streamed 38.6 years of video to 63,508 registered study participants using 111,231 unique IP addresses. About eight months of that period was spent on the "primary experiment," a randomized trial comparing Fugu with other algorithms: MPC, RobustMPC, Pensieve, and BBA (a summary of features is in Figure 5). This period saw a total of 314,577 streaming sessions, and 1,904,316 individual streams. An experimental-flow diagram in the standardized CONSORT format [35] is in the appendix (Figure A1).

We record client telemetry as time-series data, detailing the size and SSIM of every video chunk, the time to deliver each chunk to the client, the buffer size and rebuffering events at the client, the TCP statistics on the server, and the identity of the ABR and congestion-control schemes. A full description of the data is in Appendix B.

Metrics and statistical uncertainty. We group the time series by user stream to calculate a set of summary figures: the total time between the first and last recorded events of the stream, the startup time, the total watch time between the first and last successfully played portion of the stream, the total time the video is stalled for rebuffering, the average SSIM, and the chunk-by-chunk variation in SSIM. The ratio between "total time stalled" and "total watch time" is known as the rebuffering ratio or stall ratio, and is widely used to summarize the performance of streaming video systems [22].

We observe considerable heavy-tailed behavior in most of these statistics. Watch times are skewed (Figure 11), and while the *risk* of rebuffering is important to any ABR algorithm, actual rebuffering is a rare phenomenon. Of the 637,189 eligible streams considered for the primary analysis across all five ABR schemes, only 24,328 (4%) of those streams had *any* stalls, mirroring commercial services [22].

These skewed distributions create more room for the play of chance to corrupt the bottom-line statistics summarizing a scheme's performance—even two identical schemes will see considerable variation in average performance until a substantial amount of data is assembled. In this study, we worked to quantify the statistical uncertainty that can be attributed to the play of chance in assigning sessions to ABR algorithms. We calculate confidence intervals on rebuffering ratio with the bootstrap method [14], simulating streams drawn empirically from each scheme's observed distribution of rebuffering ratio as a function of stream duration. We calculate confidence intervals on average SSIM using the formula for weighted standard error, weighting each stream by its duration.

These practices result in substantial confidence intervals: with at least 2.5 years of data for each scheme, the width of the 95% confidence interval on a scheme's stall ratio is between $\pm 13\%$ and $\pm 21\%$ of the mean value. This is comparable to the magnitude of the total benefit reported by some academic work that used much shorter real-world experiments. Even a recent study of a Pensieve-like scheme on Facebook [24], encompassing 30 million streams, did not detect a change in rebuffering ratio outside the level of statistical noise.

We conclude that considerations of uncertainty in real-world learning and experimentation, especially given uncontrolled data from the Internet with real users, deserve further study. Strategies to import real-world data into repeatable emulators [45] or reduce their variance [26] will likely be helpful in producing robust learned networking algorithms.

4 Fugu: design and implementation

Fugu is a control algorithm for bitrate selection, designed to be feasibly trained in place (*in situ*) on a real deployment environment. It consists of a classical controller (model predictive control, the same as in MPC-HM), informed by a nonlinear predictor that can be trained with supervised learning.

Figure 6 shows Fugu's high-level design. Fugu runs on the server, making it easy to update its model and aggregate performance data across clients over time. Clients send necessary telemetry, such as buffer levels, to the server.

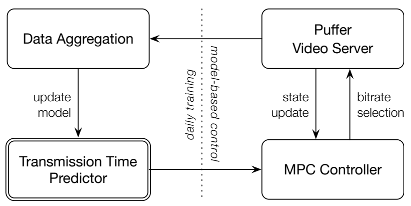


Figure 6: Overview of Fugu

The controller, described in Section 4.4, makes decisions by following a classical control algorithm to optimize an objective QoE function (§4.1) based on predictions for how long each chunk would take to transmit. These predictions are provided by the Transmission Time Predictor (TTP) (§4.2), a neural network that estimates a probability distribution for the transmission time of a proposed chunk with given size.

4.1 Objective function

For each video chunk K_i, Fugu has a selection of versions of this chunk to choose from, K_i^s, each with a different size s. As with prior approaches, Fugu quantifies the QoE of each chunk as a linear combination of video quality, video quality variation, and stall time [46]. Unlike some prior approaches, which use the average compressed bitrate of each encoding setting as a proxy for image quality, Fugu optimizes a perceptual measure of picture quality—in our case, SSIM. This has been shown to correlate with human opinions of QoE [12]. We emphasize that we use the exact same objective function in our version of MPC and RobustMPC as well.

Let $Q(K)$ be the video quality of a chunk K, $T(K)$ be the uncertain transmission time of K, and B_i be the current playback buffer size. Following [46], Fugu defines the QoE obtained by sending K_i^s (given the previously sent chunk K_{i-1}) as

$$QoE(K_i^s, K_{i-1}) = Q(K_i^s) - \lambda|Q(K_i^s) - Q(K_{i-1})| \\ - \mu \cdot \max\{T(K_i^s) - B_i, 0\}, \tag{1}$$

where $\max\{T(K_i^s) - B_i, 0\}$ describes the stall time experienced by sending K_i^s, and λ and μ are configuration constants for how much to weight video quality variation and rebuffering. Fugu plans a trajectory of sizes s of the future H chunks to maximize their expected total QoE.

4.2 Transmission Time Predictor (TTP)

Once Fugu decides which chunk from K_i^s to send, two portions of the QoE become known: the video quality and video quality variation. The remaining uncertainty is the stall time. The server knows the current playback buffer size, so what it needs to know is the transmission time: how long will it take for the client to receive the chunk? Given an oracle that reports the transmission time of any chunk, the MPC controller can compute the optimal plan to maximize QoE.

Fugu uses a trained neural-network transmission-time predictor to approximate the oracle. For each chunk in the fixed H-step horizon, we train a separate predictor. E.g., if optimizing for the total QoE of the next five chunks, five neural networks are trained. This lets us parallelize training.

Each TTP network for the future step $h \in \{0, \ldots, H-1\}$ takes as input a vector of:

1. sizes of past t chunks $K_{i-t}, \ldots, K_{i-1}$,
2. actual transmission times of past t chunks: $T_{i-t}, \ldots, T_{i-1}$,

3. internal TCP statistics (Linux `tcp_info` structure),
4. size s of a proposed chunk K^s_{i+h}.

The TCP statistics include the current congestion window size, the number of unacknowledged packets in flight, the smoothed RTT estimate, the minimum RTT, and the TCP estimated throughput (`tcpi_delivery_rate`).

Prior approaches have used Harmonic Mean (HM) [46] or a Hidden Markov Model (HMM) [40] to predict a single throughput for the entire lookahead horizon irrespective of the size of chunk to send. In contrast, the TTP acknowledges the fact that observed throughput varies with chunk size [7,32,47] by taking the size of proposed chunk K^s_{i+h} as an explicit input. In addition, it outputs a discretized probability distribution of predicted transmission time $\hat{T}(K^s_{i+h})$.

4.3 Training the TTP

We sample from the real usage data collected by *any* scheme running on Puffer and feed individual user streams to the TTP as training input. For the TTP network in the future step h, each user stream contains a chunk-by-chunk series of (a) the input 4-vector with the last element to be size of the actually sent chunk K_{i+h}, and, (b) the actual transmission time T_{i+h} of chunk K_{i+h} as desired output; the sequence is shuffled to remove correlation. It is worth noting that unlike prior work [25, 40] that learned from throughput traces, TTP is trained directly on real chunk-by-chunk data.

We train the TTP with standard supervised learning: the training minimizes the cross-entropy loss between the output probability distribution and the discretized actual transmission time using stochastic gradient descent.

We retrain the TTP every day, using training data collected over the prior 14 days, to avoid the effects of dataset shift and catastrophic forgetting [33,34]. Within the 14-day window, we weight more recent days more heavily. The weights from the previous day's model are loaded to warm-start the retraining.

4.4 Model-based controller

Our MPC controller (used for MPC-HM, RobustMPC-HM, and Fugu) is a stochastic optimal controller that maximizes the expected cumulative QoE in Equation 1 with replanning. It queries TTP for predictions of transmission time and outputs a plan $K^s_i, K^s_{i+1}, \dots, K^s_{i+H-1}$ by value iteration [8]. After sending K^s_i, the controller observes and updates the input vector passed into TTP, and replans again for the next chunk.

Given the current playback buffer level B_i and the last sent chunk K_{i-1}, let $v^*_i(B_i, K_{i-1})$ denote the maximum expected sum of QoE that can be achieved in the H-step lookahead horizon. We have value iteration as follows:

$$v^*_i(B_i, K_{i-1}) = \max_{K^s_i} \Big\{ \sum_{t_i} \Pr[\hat{T}(K^s_i) = t_i] \cdot \\ \big(QoE(K^s_i, K_{i-1}) + v^*_{i+1}(B_{i+1}, K^s_i)\big) \Big\},$$

where $\Pr[\hat{T}(K^s_i) = t_i]$ is the probability predicted by TTP for the transmission time of K^s_i to be t_i, and B_{i+1} can be derived by system dynamics [46] given an enumerated (discretized) t_i. The controller computes the optimal trajectory by solving the above value iteration with dynamic programming (DP). To make the DP computational feasible, it also discretizes B_i into bins and uses forward recursion with memoization to only compute for relevant states.

4.5 Implementation

TTP takes as input the past $t = 8$ chunks, and outputs a probability distribution over 21 bins of transmission time: $[0, 0.25), [0.25, 0.75), [0.75, 1.25), \dots, [9.75, \infty)$, with 0.5 seconds as the bin size except for the first and the last bins. TTP is a fully connected neural network, with two hidden layers with 64 neurons each. We tested different TTPs with various numbers of hidden layers and neurons, and found similar training losses across a range of conditions for each. We implemented TTP and the training in PyTorch, but we load the trained model in C++ when running on the production server for performance. A forward pass of TTP's neural network in C++ imposes minimal overhead per chunk (less than 0.3 ms on average on a recent x86-64 core). The MPC controller optimizes over $H = 5$ future steps (about 10 seconds). We set $\lambda = 1$ and $\mu = 100$ to balance the conflicting goals in QoE. Each retraining takes about 6 hours on a 48-core server.

4.6 Ablation study of TTP features

We performed an ablation study to assess the impact of the TTP's features (Figure 7). The prediction accuracy is measured using mean squared error (MSE) between the predicted transmission time and the actual (absolute, unbinned) value. For the TTP that outputs a probability distribution, we compute the expected transmission time by weighting the median value of each bin with the corresponding probability. Here are the more notable results:

Use of low-level congestion-control statistics. The TTP's nature as a DNN lets it consider a variety of noisy inputs, including low-level congestion-control statistics. We feed the kernel's `tcp_info` structure to the TTP, and find that several of these fields contribute positively to the TTP's accuracy, especially the RTT, CWND, and number of packets in flight (Figure 7). Although client-side ABR systems cannot typically access this structure directory because the statistics live on the sender, these results should motivate the communication of richer data to ABR algorithms wherever they live.

Transmission-time prediction. The TTP explicitly considers the size of a proposed chunk, rather than predicting throughput and then modeling transmission time as scaling linearly with chunk size [7, 32, 47]. We compared the TTP with an equivalent throughput predictor that is agnostic to the

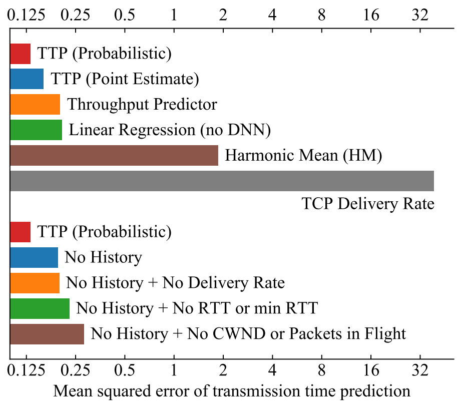


Figure 7: Ablation study of Fugu's Transmission Time Predictor (TTP). Removing any of the TTP's inputs reduced its ability to predict the transmission time of a video chunk. A non-probabilistic TTP ("Point Estimate") and one that predicts throughput without regard to chunk size ("Throughput Predictor") both performed markedly worse. TCP statistics (RTT, CWND, packets in flight) also proved helpful.

chunk's size (keeping everything else unchanged). The TTP's predictions were much more accurate (Figure 7).

Prediction with uncertainty. The TTP outputs a *probability distribution* of transmission times. This allows for better decision making compared with a single point estimate without uncertainty. We evaluated the expected accuracy of a probabilistic TTP vs. a point-estimate version that outputs the median value of the most-probable bin, and found an improvement in prediction accuracy with the former (Figure 7). To measure the end-to-end benefits of a probabilistic TTP, we deployed both versions on Puffer in August 2019 and collected 39 stream-days of data. It performed much worse than normal Fugu: the rebuffering ratio was 5× worse, without significant improvement in SSIM (data not shown).

Use of neural network. We found a significant benefit from using a deep neural network in this application, compared with a linear-regression model that was trained the same way. The latter model performed much worse on prediction accuracy (Figure 7). We also deployed it on Puffer and collected 448 stream-days of data in Aug.–Oct. 2019; its rebuffering ratio was 2.5× worse (data not shown).

Daily retraining. To evaluate our practice of retraining the TTP each day, we conducted a randomized comparison of several "out-of-date" versions of the TTP on Puffer between Aug. 7 and Aug. 30, 2019, and between Oct. 16, 2019 and Jan. 2, 2020. We compared vintages of the TTP that had been trained in February, March, April, and May 2019, alongside the TTP that is retrained each day. (We emphasize that the older TTP vintages were also learned *in situ* on two weeks of data from the actual deployment environment—they are simply earlier versions of the same predictor.) Somewhat to our

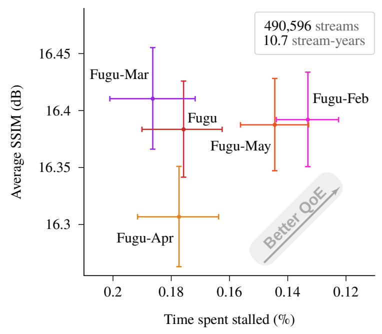


Figure 8: Fugu, which is retrained every day, did not outperform older versions of itself that were trained up to 11 months earlier. Our practice of daily retraining appears to be overkill.

surprise and disappointment, we were not able to document a benefit from daily retraining (Figure 8). This may reflect a lack of dynamism in the Puffer userbase, or the fact that once "enough" data is available to put the predictor through its paces, more-recent data is not necessarily beneficial, or some other reason. We suspect the older predictors might become stale at some point in the future, but for the moment, our practice of daily retraining appears to be overkill.

5 Experimental results

We now present findings from our experiments with the Puffer study, including the evaluation of Fugu. Our main results are shown in Figure 9. In summary, we conducted a parallel-group, blinded-assignment, randomized controlled trial of five ABR schemes between Jan. 26 and Aug. 7, and between Aug. 30 and Oct. 16, 2019. The data include 13.1 stream-years of data split across five algorithms, counting all streams that played at least 4 seconds of video. A standardized diagram of the experimental flow is available in the appendix (Figure A1).

We found that simple "buffer-based" control (BBA) performs surprisingly well, despite its status as a frequently outperformed research baseline. The only scheme to consistently outperform BBA in both stalls and quality was Fugu, but only when *all* features of the TTP were used. If we remove the probabilistic "fuzzy" nature of Fugu's predictions, *or* the "depth" of the neural network, *or* the prediction of transmission time as a function of chunk size (and not simply throughput), Fugu forfeits its advantage (§4.6). Fugu also outperformed other schemes in terms of SSIM variability (Figure 1). On a cold start to a new session, prior work [19, 40] suggested a need for session clustering to determine the quality of the first chunk. TTP provides an alternative approach: low-level TCP statistics are available as soon as the (HTTP/WebSocket, TLS, TCP) connection is established and allow Fugu to begin safely at a higher quality (Figure 10).

We conclude that robustly beating "simple" algorithms with machine learning may be surprisingly difficult, notwith-

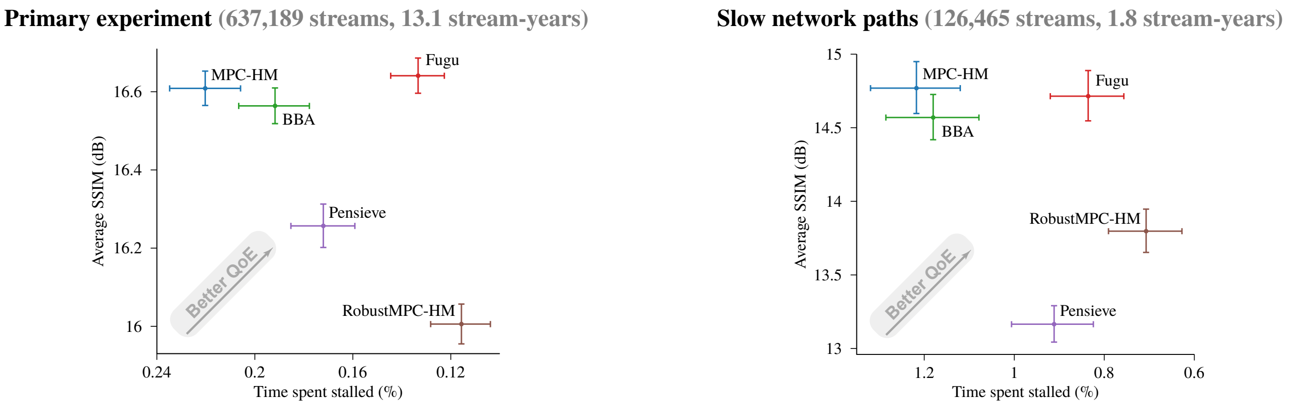


Figure 9: Main results. In a blinded randomized controlled trial that included 13.1 years of video streamed to 54,612 client IP addresses over an eight-month period, Fugu reduced the fraction of time spent stalled (except with respect to RobustMPC-HM), increased SSIM, and reduced SSIM variation within each stream (tabular data in Figure 1). "Slow" network paths have average throughput less than 6 Mbit/s; following prior work [25, 46], these paths are more likely to require nontrivial bitrate-adaptation logic. Such streams accounted for 14% of overall viewing time and 83% of stalls. Error bars show 95% confidence intervals.

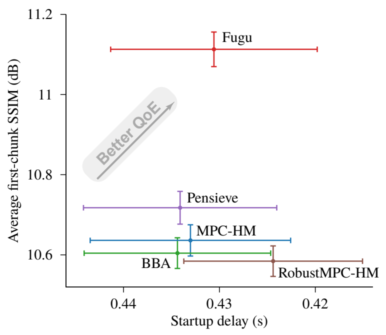


Figure 10: On a cold start, Fugu's ability to bootstrap ABR decisions from TCP statistics (e.g., RTT) boosts initial quality.

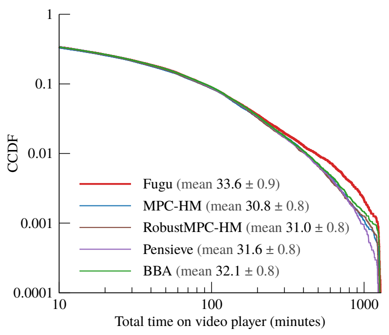


Figure 11: Users randomly assigned to Fugu chose to remain on the Puffer video player about 5%–9% longer, on average, than those assigned to other schemes. Users were blinded to the assignment. Legend shows 95% confidence intervals on the average time-on-site in minutes.

standing promising results in contained environments such as simulators and emulators. The gains that learned algorithms have in optimization or smarter decision making may come at a tradeoff in brittleness or sensitivity to heavy-tailed behavior.

5.1 Fugu users streamed for longer

We observed significant differences in the session durations of users across algorithms (Figure 11). Users whose sessions were assigned to Fugu chose to remain on the Puffer video player about 5–9% longer, on average, than those assigned to other schemes. Users were blinded to the assignment, and we believe the experiment was carefully executed not to "leak" details of the underlying scheme (MPC and Fugu even share most of their codebase). The average difference was driven solely by the upper 4% tail of viewership duration (sessions lasting more than 3 hours)—viewers assigned to Fugu are much more likely to keep streaming beyond this point, even as the distributions are nearly identical until then.

Time-on-site is a figure of merit in the video-streaming industry and might be increased by delivering better-quality video with fewer stalls, but we simply do not know enough about what is driving this phenomenon.

5.2 The benefits of learning *in situ*

Each of the ABR algorithms we deployed has been evaluated in emulation in prior work [25, 46]. Notably, the results in those works are qualitatively different from some of the real world results we have seen here—for example, buffer-based control matching or outperforming MPC-HM and Pensieve.

To investigate this further, we constructed an emulation environment similar to that used in [25]. This involved running the Puffer media server locally, and launching headless Chrome clients inside mahimahi [30] shells to connect to the server. Each mahimahi shell imposed a 40 ms end-to-end delay on traffic originating inside it and limited the downlink

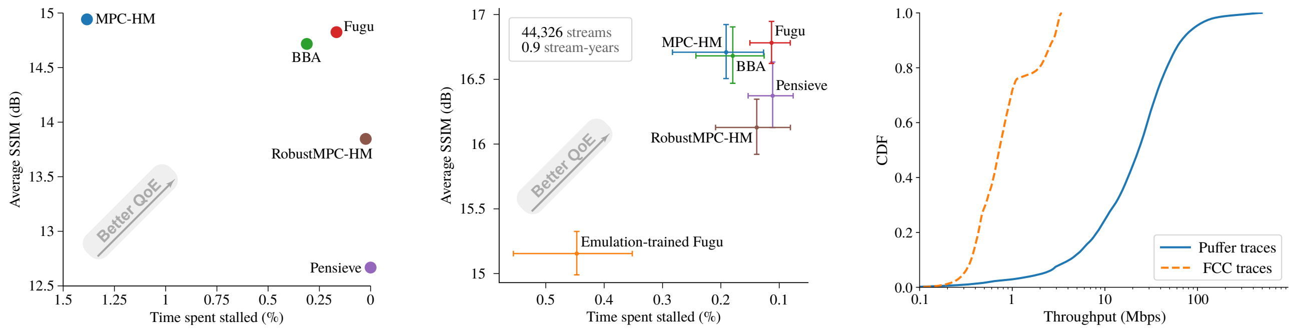


Figure 12: Left: performance in emulation, run in mahimahi [30] using the FCC traces [10], following the method of Pensieve [25]. **Middle:** During Jan. 26–Apr. 2, 2019, we randomized sessions to a set of algorithms including "emulation-trained Fugu." For Fugu, training in emulation did not generalize to the deployment environment. In addition, emulation results (left) are not indicative of real-world performance. **Right:** comparison of throughput distribution of FCC traces and of real Puffer sessions.

capacity over time to match the capacity recorded in a set of FCC broadband network traces [10]. As in the Pensieve evaluation, uplink speeds in all shells were capped at 12 Mbps. Within this test setup, we automated 12 clients to repeatedly connect to the media server, which would play a 10 minute clip recorded on NBC over each network trace in the dataset. Each client was assigned to a different ABR algorithm, and played the 10 minute video repeatedly over more than 15 hours of FCC traces. Results are shown in Figure 12.

We trained a version of Fugu in this emulation environment to evaluate its performance. Compared with the *in situ* Fugu—or with every other ABR scheme—the real-world performance of emulation-trained Fugu was horrible (Figure 12, middle panel). Looking at the other ABR schemes, almost each of them lies somewhere along the SSIM/stall frontier in emulation (left side of figure), with Pensieve rebuffering the least and MPC delivering the highest quality video. In the real experiment (middle of figure), we see a more muddled picture, with a different qualitative arrangement of schemes.

5.3 Remarks on Pensieve and RL for ABR

The original Pensieve paper [25] demonstrated that Pensieve outperformed MPC-HM, RobustMPC-HM, and BBA in both emulation-based tests and in video streaming tests on low and high-speed real-world networks. Our results differ; we believe the mismatch may have occurred for several reasons.

First, we have found that simulation-based training and testing do not capture the vagaries of the real-world paths seen in the Puffer study. Unlike real-world randomized trials, trace-based emulators and simulators allow experimenters to limit statistical uncertainty by running different algorithms on the same conditions, eliminating the effect of the play of chance in giving different algorithms a different distribution of watch times, network behaviors, etc. However, it is difficult to characterize the *systematic* uncertainty that comes from selecting a set of traces that may omit the variability or heavy-tailed nature of a real deployment experience (both network behaviors as well as user behaviors, such as watch duration).

Reinforcement learning (RL) schemes such as Pensieve may be at a particular disadvantage from this phenomenon. Unlike supervised learning schemes that can learn from training "data," RL typically requires a training *environment* to respond to a sequence of control decisions and decide on the appropriate consequences and reward. That environment could be real life instead of a simulator, but the level of statistical noise we observe would make this type of learning extremely slow or require an extremely broad deployment of algorithms in training. RL relies on being able to slightly vary a control action and detect a change in the resulting reward. By our calculations, the variability of inputs is such that it takes about 2 stream-years of data to reliably distinguish two ABR schemes whose innate "true" performance differs by 15%. To make RL practical, future work may need to explore techniques to reduce this variability [26] or construct more faithful simulators and emulators that model tail behaviors and capture additional dynamics of the real Internet that are not represented in throughput traces (e.g. varying RTT, cross traffic, interaction between throughput and chunk size [7]).

Second, most of the evaluation of Pensieve in the original paper focused on training and evaluating Pensieve using a single test video. As a result, the state space that model had to explore was inherently more limited. Evaluation of the Pensieve "multi-video model"—which we have to use for our experimental setting—was more limited. Our results are more consistent with a recent large-scale study of a Pensieve-multi-video-like scheme on 30 million streams at Facebook [24].

Third, the right side of Figure 12 shows that the distribution of throughputs in the FCC traces differs markedly from those on Puffer. This dataset shift could have harmed the performance of Pensieve, which was trained on the FCC traces. In response to reviewer feedback, we trained a version of Pensieve on throughput traces randomly sampled from real Puffer video sessions. This is essentially as close to a "learned *in situ*" version of Pensieve as we think we can achieve, but is

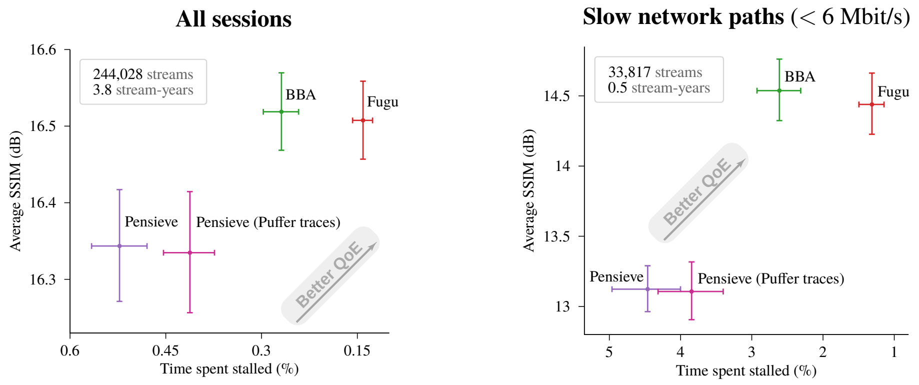


Figure 13: During Jan. 2–Feb. 2, 2020, we evaluated a version of Pensieve that was trained on a collection of network traces drawn randomly from actual Puffer sessions. This improved its performance compared with the original Pensieve, but the overall results were broadly similar.

not quite the same (§5.3). We compared "Pensieve on Puffer traces" with the original Pensieve, BBA, and Fugu between Jan. 2 and Feb. 2, 2020 (Figure 13). The results were broadly similar; the new Pensieve achieved better performance, but was still significantly worse than BBA and Fugu. The results deserve further study; they suggest that the representativeness of training data is not the end of the story when it comes to the real-world performance of RL schemes trained in simulation.

Finally, Pensieve optimizes a QoE metric centered around bitrate as a proxy for video quality. We did not alter this and leave the discussion to Section 6. Figure 4 shows that Pensieve was the #2 scheme in terms of bitrate (below BBA) in the primary analysis. We emphasize that our findings do not indicate that Pensieve cannot be a useful ABR algorithm, especially in a scenario where similar, pre-recorded video is played over a familiar set of known networks.

6 Limitations

The design of the Puffer experiment and the Fugu system are subject to important limitations that may affect their performance and generalizability.

6.1 Limitations of the experiments

Our randomized controlled trial represents a rigorous, but necessarily "black box," study of ABR algorithms for video streaming. We don't know the true distribution of network paths and throughput-generating processes; we don't know the participants or why the distribution in watch times differs by assigned algorithm; we don't know how to emulate these behaviors accurately in a controlled environment.

We have supplemented this black-box work with ablation analyses to relate the real-world performance of Fugu to the l^2 accuracy of its predictor, and have studied various ablated versions of Fugu in deployment. However, ultimately part of the reason for this paper is that we *cannot* replicate the experimental findings outside the real world—a real world whose behavior is noisy and takes lots of time to measure precisely. That may be an unsatisfying conclusion, and we doubt it will be the final word on this topic. Perhaps it will become possible to model enough of the vagaries of the real Internet "in silico" to enable the development of robust control strategies without extensive real-world experiments.

It is also unknown to what degree Puffer's results—which are about a single server in a university datacenter, sending to clients across our entire country over the wide-area Internet—generalize to a different server at a different institution, much less the more typical paths between a user on an access network and their nearest CDN edge node. We don't know for sure if the pre-trained Fugu model would work in a different location, or whether training a new Fugu based on data from that location would yield comparable results. Our results show that learning *in situ* works, but we don't know how specific the *situs* needs to be. And while we expect that Fugu could be implemented in the context of client-side ABR (especially if the server is willing to share its `tcp_info` statistics with the client), we haven't demonstrated this.

Although we believe that past research papers may have underestimated the uncertainties in real-world measurements with realistic Internet paths and users, we also may be guilty of underestimating our own uncertainties or emphasizing uncertainties that are only relevant to small or medium-sized academic studies, such as ours, and irrelevant to the industry. The current load on Puffer is about 60 concurrent streams on average, meaning we collect about 60 stream-days of data per day. Our primary analysis covers about 2.6 stream-years of data per scheme collected over an eight-month period, and was sufficient to measure its performance metrics to within about $\pm 15\%$ (95% CI). By contrast, we understand YouTube has an average load of more than 60 *million* concurrent streams at any given time. We imagine the considerations of conducting data-driven experiments at this level may be completely different—perhaps less about statistical uncertainty, and more about systematic uncertainties and the difficulties of running experiments and accumulating so much data.

Some of Fugu's performance (and that of MPC, RobustMPC, and BBA) relative to Pensieve may be due to the fact that these four schemes received more information as they ran—namely, the SSIM of each possible version of each future chunk—than did Pensieve. It is possible that an "SSIM-aware" Pensieve might perform better. The load of calculating SSIM for each encoded chunk is not insignificant—about an extra 40% on top of encoding the video.

6.2 Limitations of Fugu

There is a sense that data-driven algorithms that more "heavily" squeeze out performance gains may also put themselves at risk to brittleness when a deployment environment drifts from one where the algorithm was trained. In that sense, it is hard to say whether Fugu's performance might decay catastrophically some day. We tried and failed to demonstrate a quantitative benefit from daily retraining over "out-of-date" vintages, but at the same time, we cannot be sure that some surprising detail tomorrow—e.g., a new user from an unfamiliar network—won't send Fugu into a tailspin before it can be retrained. A year of data on a growing userbase suggests, but doesn't guarantee, robustness to a changing environment.

Fugu does not consider several issues that other research has concerned itself with—e.g., being able to "replace" already-downloaded chunks in the buffer with higher quality versions [38], or optimizing the joint QoE of multiple clients who share a congestion bottleneck [29].

Fugu is not tied as tightly to the TCP or congestion control as it might be—for example, Fugu could wait to send a chunk until the TCP sender tells it that there is a sufficient congestion window for most of the chunk (or the whole chunk) to be sent immediately. Otherwise, it *might* choose to wait and make a better-informed decision later. Fugu does not schedule the transmission of chunks—it will always send the next chunk as long as the client has room in its playback buffer.

7 Conclusion

Machine-learned systems in computer networking sometimes describe themselves as achieving near-"optimal" performance, based on results in a contained or modeled version of the problem [25, 37, 39]. Such approaches are not limited to the academic community: in early 2020, a major video-streaming company announced a $5,000 prize for the best low-delay ABR scheme, in which candidates will be evaluated in a network simulator that follows a trace of varying throughput [2].

In this paper, we suggest that these efforts can benefit from considering a broader notion of performance and optimality. Good, or even near-optimal, performance in a simulator or emulator does not necessarily predict good performance over the wild Internet, with its variability and heavy-tailed distributions. It remains a challenging problem to gather the appropriate training data (or in the case of RL systems, training environments) to properly learn and validate such systems.

In this paper, we asked: *what does it take to create a learned ABR algorithm that robustly performs well over the wild Internet?* In effect, our best answer is to cheat: train the algorithm *in situ* on data from the real deployment environment, and use an algorithm whose structure is sophisticated enough (a neural network) and yet also simple enough (a predictor amenable to supervised learning on data, informing a classical controller) to benefit from that kind of training.

Over the last year, we have streamed 38.6 years of video to 63,508 users across the Internet. Sessions are randomized in blinded fashion among algorithms, and client telemetry is recorded for analysis. The Fugu algorithm robustly outperformed other schemes, both simple and sophisticated, on objective measures (SSIM, stall time, SSIM variability) and increased the duration that users chose to continue streaming.

We have found the Puffer approach a powerful tool for networking research—it is fulfilling to be able to "measure, then build" [5] to iterate rapidly on new ideas and gain feedback. Accordingly, we are opening Puffer as an "open research" platform. Along with this paper, we are publishing our full archive of data and results on the Puffer website. The system posts new data each week, along with a summary of results from the ongoing experiments, with confidence intervals similar to those in this paper. (The format is described in Appendix B.) We redacted some fields from the public archive to protect participants' privacy (e.g., IP address) but are willing to work with researchers on access to these fields in an aggregated fashion. Puffer and Fugu are also open-source software, as are the analysis tools used to prepare the results in this paper.

We plan to operate Puffer as long as feasible and invite researchers to train and validate new algorithms for ABR control, network and throughput prediction, and congestion control on its traffic. We are eager to collaborate with and learn from the community's ideas on how to design and deploy robust learned systems for the Internet.

Acknowledgments

We are greatly indebted to Emily Marx, who joined this project after the original submission of this paper, found and corrected bugs in our analysis tools, and performed the final data analysis. We thank our shepherd, Vyas Sekar, and the ACM SIGCOMM and USENIX NSDI reviewers for their helpful feedback. We are grateful for conversations with and feedback from Danfei Xu, T.Y. Huang, Hongzi Mao, Michael Schapira, and Nils Krahnstoever, and we thank the participants in the Puffer research study, without whom these experiments could not have been conducted. This work was supported by NSF grant CNS-1909212 and by Google, Huawei, VMware, Dropbox, Facebook, and the Stanford Platform Lab.

References

[1] Locast: Non-profit retransmission of broadcast television, June 2018. https://news.locast.org/app/uploads/2018/11/Locast-White-Paper.pdf.

[2] MMSys'20/Twitch Grand Challenge on Adaptation Algorithms for Near-Second Latency, January 2020. https://2020.acmmmsys.org/lll_challenge.php.

[3] Alekh Agarwal, Nan Jiang, and Sham M. Kakade. Lecture notes on the theory of reinforcement learning. 2019.

[4] Zahaib Akhtar, Yun Seong Nam, Ramesh Govindan, Sanjay Rao, Jessica Chen, Ethan Katz-Bassett, Bruno Ribeiro, Jibin Zhan, and Hui Zhang. Oboe: Auto-tuning video ABR algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM SIGCOMM*, pages 44–58, 2018.

[5] Remzi Arpaci-Dusseau. Measure, then build (USENIX ATC 2019 keynote). Renton, WA, July 2019. USENIX Association.

[6] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for Internet video. *ACM SIGCOMM Computer Communication Review*, 43(4):339–350, 2013.

[7] Mihovil Bartulovic, Junchen Jiang, Sivaraman Balakrishnan, Vyas Sekar, and Bruno Sinopoli. Biases in data-driven networking, and what to do about them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 192–198, 2017.

[8] Richard Bellman. A Markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[9] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14(5):20–53, 2016.

[10] Federal Communications Commission. Measuring Broadband America. https://www.fcc.gov/general/measuring-broadband-america.

[11] Paul Crews and Hudson Ayers. CS 244 '18: Recreating and extending Pensieve, 2018. https://reproducingnetworkresearch.wordpress.com/2018/07/16/cs-244-18-recreating-and-extending-pensieve/.

[12] Zhengfang Duanmu, Kai Zeng, Kede Ma, Abdul Rehman, and Zhou Wang. A quality-of-experience index for streaming video. *IEEE Journal of Selected Topics in Signal Processing*, 11(1):154–166, 2016.

[13] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. In *ICML 2019 Workshop RL4RealLife*, 2019.

[14] Bradley Efron and Robert Tibshirani. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical science*, pages 54–75, 1986.

[15] Sally Floyd and Eddie Kohler. Internet research needs better models. *ACM SIGCOMM Computer Communication Review*, 33(1):29–34, 2003.

[16] Sally Floyd and Vern Paxson. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, 2001.

[17] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, 2018.

[18] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 Conference of the ACM SIGCOMM*, pages 187–198, 2014.

[19] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. CFA: A practical prediction system for video QoE optimization. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 137–150, 2016.

[20] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE. In *Proceedings of the 8th International Conference on emerging Networking EXperiments and Technologies*, pages 97–108, 2012.

[21] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking*, 21(6):2001–2014, 2013.

[22] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the 2017 Conference of the ACM SIGCOMM*, pages 183–196, 2017.

[23] Zhi Li, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali C. Begen, and David Oran. Probe and adapt: Rate adaptation for HTTP video streaming at scale.

IEEE Journal on Selected Areas in Communications, 32(4):719–733, 2014.

[24] Hongzi Mao, Shannon Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, Mohammad Alizadeh, and Eytan Bakshy. Real-world video adaptation with reinforcement learning. In *ICML 2019 Workshop RL4RealLife*, 2019.

[25] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. In *Proceedings of the 2017 Conference of the ACM SIGCOMM*, pages 197–210. ACM, 2017.

[26] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. Variance reduction for reinforcement learning in input-driven environments. In *International Conference on Learning Representations*, 2019.

[27] Ricky K.P. Mok, Xiapu Luo, Edmond W.W. Chan, and Rocky K.C. Chang. QDASH: a QoE-aware DASH system. In *Proceedings of the 3rd Multimedia Systems Conference*, pages 11–22, 2012.

[28] *Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats*, April 2012. ISO/IEC 23009-1 (http://standards.iso.org/ittf/PubliclyAvailableStandards).

[29] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 408–423, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[31] Vern Paxson and Sally Floyd. Why we don't know how to simulate the Internet. In *Proceedings of the 29th conference on Winter simulation*, pages 1037–1044, 1997.

[32] Yanyuan Qin, Shuai Hao, Krishna R. Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. ABR streaming of VBR-encoded videos: characterization, challenges, and solutions. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 366–378. ACM, 2018.

[33] Anthony Robins. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2):123–146, 1995.

[34] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 627–635, 2011.

[35] Kenneth F. Schulz, Douglas G. Altman, and David Moher. CONSORT 2010 statement: updated guidelines for reporting parallel group randomised trials. *BMC medicine*, 8(1):18, 2010.

[36] Alexander T. Schwarm and Michael Nikolaou. Chance-constrained model predictive control. *AIChE Journal*, 45(8):1743–1752, 1999.

[37] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An experimental study of the learnability of congestion control. In *Proceedings of the 2014 Conference of the ACM SIGCOMM*, pages 479–490, 2014.

[38] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From theory to practice: Improving bitrate adaptation in the DASH reference player. In *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, pages 123–137, New York, NY, USA, 2018. ACM.

[39] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. BOLA: Near-optimal bitrate adaptation for online videos. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.

[40] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. CS2P: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 Conference of the ACM SIGCOMM*, pages 272–285, 2016.

[41] Cisco Systems. Cisco Visual Networking Index: Forecast and trends, 2017–2022, November 2018. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.pdf.

[42] Guibin Tian and Yong Liu. Towards agile and smooth video adaptation in dynamic HTTP streaming. In *Proceedings of the 8th International Conference on emerging Networking EXperiments and Technologies*, pages 109–120, 2012.

[43] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

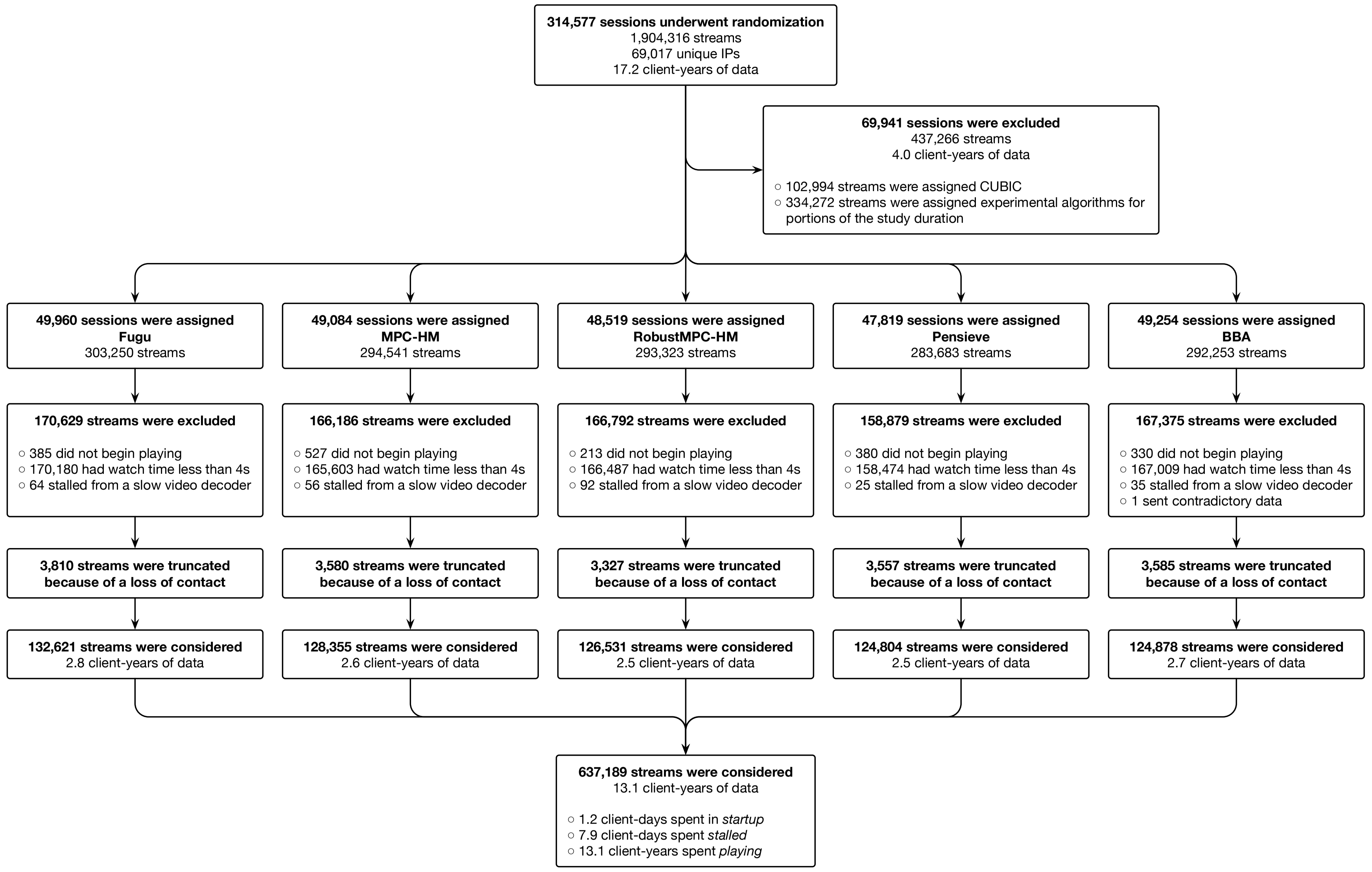
[44] Keith Winstein and Hari Balakrishnan. TCP ex Machina: Computer-generated congestion control. *Proceedings of the 2013 Conference of the ACM SIGCOMM*, 43(4):123–134, 2013.

[45] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for Internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, Boston, MA, 2018. USENIX Association.

[46] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proceedings of the 2015 Conference of the ACM SIGCOMM*, pages 325–338, 2015.

[47] Tong Zhang, Fengyuan Ren, Wenxue Cheng, Xiaohui Luo, Ran Shu, and Xiaolan Liu. Modeling and analyzing the influence of chunk size variation on bitrate adaptation in DASH. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

A Randomized trial flow diagram



314,577 sessions underwent randomization
1,904,316 streams
69,017 unique IPs
17.2 client-years of data
69,941 sessions were excluded
437,266 streams
4.0 client-years of data
○ 102,994 streams were assigned CUBIC
○ 334,272 streams were assigned experimental algorithms for portions of the study duration
49,960 sessions were assigned Fugu
303,250 streams
49,084 sessions were assigned MPC-HM
294,541 streams
48,519 sessions were assigned RobustMPC-HM
293,323 streams
47,819 sessions were assigned Pensieve
283,683 streams
49,254 sessions were assigned BBA
292,253 streams
170,629 streams were excluded
○ 385 did not begin playing
○ 170,180 had watch time less than 4s
○ 64 stalled from a slow video decoder
166,186 streams were excluded
○ 527 did not begin playing
○ 165,603 had watch time less than 4s
○ 56 stalled from a slow video decoder
166,792 streams were excluded
○ 213 did not begin playing
○ 166,487 had watch time less than 4s
○ 92 stalled from a slow video decoder
158,879 streams were excluded
○ 380 did not begin playing
○ 158,474 had watch time less than 4s
○ 25 stalled from a slow video decoder
167,375 streams were excluded
○ 330 did not begin playing
○ 167,009 had watch time less than 4s
○ 35 stalled from a slow video decoder
○ 1 sent contradictory data
3,810 streams were truncated because of a loss of contact
3,580 streams were truncated because of a loss of contact
3,327 streams were truncated because of a loss of contact
3,557 streams were truncated because of a loss of contact
3,585 streams were truncated because of a loss of contact
132,621 streams were considered
2.8 client-years of data
128,355 streams were considered
2.6 client-years of data
126,531 streams were considered
2.5 client-years of data
124,804 streams were considered
2.5 client-years of data
124,878 streams were considered
2.7 client-years of data
637,189 streams were considered
13.1 client-years of data
○ 1.2 client-days spent in *startup*
○ 7.9 client-days spent *stalled*
○ 13.1 client-years spent *playing*

Figure A1: CONSORT-style diagram [35] of experimental flow for the primary results (Figures 1 and 9), obtained during the period Jan. 26–Aug. 7, 2019, and Aug. 30–Oct. 16, 2019. A "session" represents one visit to the Puffer video player and may contain many "streams." Reloading starts a new session, but changing channels only starts a new stream and does not change TCP connections or ABR algorithms.

B Description of open data

The open data we are releasing comprise different "measurements"—each measurement contains a different set of time-series data collected on Puffer servers. Below we highlight the format of interesting fields in three measurements that are essential for analysis: `video_sent`, `video_acked`, and `client_buffer`.

`video_sent` collects a data point every time a Puffer server sends a video chunk to a client. Each data point contains:

- `time`: timestamp when the chunk is sent
- `session_id`: unique ID for the video session
- `expt_id`: unique ID to identify the experimental group; `expt_id` can be used as a key to retrieve the experimental setting (e.g., ABR, congestion control) when sending the chunk, in another file we are providing.
- `channel`: TV channel name
- `video_ts`: unique presentation timestamp of the chunk
- `format`: encoding settings of the chunk, including resolution and constant rate factor (CRF)
- `size`: size of the chunk
- `ssim_index`: SSIM of the chunk
- `cwnd`: congestion window size (`tcpi_snd_cwnd`)
- `in_flight`: number of unacknowledged packets in flight (`tcpi_unacked - tcpi_sacked - tcpi_lost + tcpi_retrans`)
- `min_rtt`: minimum RTT (`tcpi_min_rtt`)
- `rtt`: smoothed RTT estimate (`tcpi_rtt`)
- `delivery_rate`: estimate of TCP throughput (`tcpi_delivery_rate`)

`video_acked` collects a data point every time a Puffer server receives a video chunk acknowledgement from a client. Each data point can be matched to a data point in `video_sent` using `video_ts` (if the chunk is ever acknowledged) and used to calculate the transmission time of the chunk—difference between the timestamps in the two data points. Specifically, each data point in `video_acked` contains:

- `time`: timestamp when the chunk is acknowledged
- `session_id`
- `expt_id`
- `channel`
- `video_ts`

`client_buffer` collects client-side information reported to Puffer servers on a regular interval and when certain events occur. Each data point contains:

- `time`: timestamp when the client message is received
- `session_id`
- `expt_id`
- `channel`
- `event`: event type, e.g., was this triggered by a regular report every quarter second, or because the client stalled or began playing.
- `buffer`: playback buffer size
- `cum_rebuf`: cumulative rebuffer time in the current stream

Between Jan. 26, 2019 and Feb. 2, 2020, we collected 675,839,652 data points in `video_sent`, 677,956,279 data points in `video_acked`, and 4,622,575,336 data points in `client_buffer`.

Is Big Data Performance Reproducible in Modern Cloud Networks?

Alexandru Uta[1], *Alexandru Custura*[1], *Dmitry Duplyakin*[2], *Ivo Jimenez*[3], *Jan Rellermeyer*[4], *Carlos Maltzahn*[3], *Robert Ricci*[2], *Alexandru Iosup*[1]
[1]VU Amsterdam [2]University of Utah [3]University of California, Santa Cruz [4]TU Delft

Abstract

Performance variability has been acknowledged as a problem for over a decade by cloud practitioners and performance engineers. Yet, our survey of top systems conferences reveals that the research community regularly disregards variability when running experiments in the cloud. Focusing on networks, we assess the impact of variability on cloud-based big-data workloads by gathering traces from mainstream commercial clouds and private research clouds. Our dataset consists of millions of datapoints gathered while transferring over 9 petabytes on cloud providers' networks. We characterize the network variability present in our data and show that, even though commercial cloud providers implement mechanisms for quality-of-service enforcement, variability still occurs, and is even exacerbated by such mechanisms and service provider policies. We show how big-data workloads suffer from significant slowdowns and lack predictability and replicability, even when state-of-the-art experimentation techniques are used. We provide guidelines to reduce the volatility of big data performance, making experiments more repeatable.

1 Introduction

Performance variability [13, 47] in the cloud is well-known, and has been studied since the early days [7, 35, 55] of cloud computing. Cloud performance variability impacts not only operational concerns, such as cost and predictability [14, 42], but also reproducible experiment design [3, 10, 31, 47].

Big data is now highly embedded in the cloud: for example, Hadoop [64] and Spark [65] processing engines have been deployed for many years on on-demand resources. One key issue when running big data workloads in the cloud is that, due to the multi-tenant nature of clouds, applications see performance effects from other tenants, and are thus susceptible to performance variability, including on the network. Even though recent evidence [50] suggests that there are limited potential gains from *speeding up* the network, it is still the case that variable network performance can *slow down* big data systems and introduce volatility that makes it more difficult to draw reliable scientific conclusions.

Although cloud performance variability has been thoroughly studied, the resulting work has mostly been in the context of optimizing tail latency [22], with the aim of providing more consistent application-level performance [15, 25, 29, 57]. This is subtly—but importantly—different from understanding the ways that fine-grained, resource-level variability affects the *performance evaluation* of these systems. Application-level effects are especially elusive for complex applications, such as big data, which are not bottlenecked on a specific resource for their entire runtime. As a result, it is difficult for experimenters to understand how to design experiments that lead to reliable conclusions about application performance under variable network conditions.

Modern cloud data centers increasingly rely on software-defined networking to offer flows between VMs with reliable and predictable performance [48]. While modern cloud networks generally promise isolation and predictability [7, 30], in this paper we uncover that they rarely achieve stable performance. Even the mechanisms and policies employed by cloud providers for offering quality of service (QoS) and fairness can result in non-trivial interactions with the user applications, which leads to performance variability.

Although scientists are generally aware of the relationship between repeated experiments and increased confidence in results, the specific strength of these effects, their underlying causes, and methods for improving experiment designs have not been carefully studied in the context of performance experiments run in clouds. Variability has a significant impact on sound experiment design and result reporting [31]. In the presence of variability, large numbers of experiment repetitions must be performed to achieve tight confidence intervals [47]. Although practitioners and performance engineers acknowledge this phenomenon [7, 35, 55], in practice these effects are frequently disregarded in performance studies.

Building on our vision [34], and recognizing the trend of the academic community's increasing use of the cloud for computing resources [53], we challenge the current state-of-

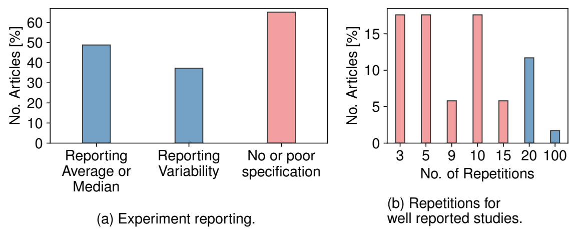


Figure 1: State-of-practice in big data articles with cloud experiments: (a) Aspects reported about experiments. Bars represent aspects that are not mutually exclusive, thus the total can exceed 100%. (b) Number of experiment repetitions performed for the properly specified articles.

practice in cloud-based systems experimentation and advocate for sound experiment design and result reporting. We show that, due to performance variability, flawed cloud-based experimentation could lead to inaccurate or even wrong conclusions. We show, in-depth, the performance implications of network variability when running big data workloads. The interplay between underlying resources and applications is complex, and leads to non-trivial performance behavior.

To characterize such interactions, we run state-of-the-art, real-world applications using Apache Spark [4]. We run big-data workloads either directly on real-world mainstream clouds, or by emulating the network behavior of such clouds. Our results show that variability highly impacts not only performance, but also credible and reproducible experimentation.

Addressing cloud users, performance engineers, and system designers, we examine the implications of *network* variability on big data, and present our main findings and contributions:

1. **Lack of sound experimentation:** Many articles in the literature that present cloud-based experiments are either under-specified (i.e., do not report statistical measures), or run inconclusive numbers of experiment repetitions (Section 2).
2. **Variability in modern cloud networks:** We conduct and analyze measurements of public and private cloud providers, characterize the level of variability, and identify specific sources (Section 3).
3. **Network variability impact on application performance reproducibility:** Low-level network variability can have significant effects on application performance, and can violate assumptions commonly used in performance modeling (such as that experiment runs are independent and identically distributed) (Section 4).
4. **Strategies for running reproducible experiments:** Given our measurement and experience with application-level benchmarks, we make recommendations for improving the reliability and reproducibility of experiments (Section 5).

2 Is Cloud Variability Disregarded?

We perform a literature survey to uncover whether and how researchers and practitioners take cloud performance variability into account when running experiments. Our findings are depicted in Figure 1 and summarized as follows:

Table 1: Parameters for the performance variability literature survey. We manually select only the articles with empirical evaluations performed using clouds.

Venues	Keywords	Years
NSDI, OSDI SOSP, SC	big data, streaming, Hadoop, MapReduce, Spark, data storage graph processing, data analytics	2008 - 2018

Table 2: Survey process. Initial filtering done automatically by keywords, then manually for cloud-based experiments. The resulting subset is significant and highly-cited.

Articles Total	Filtered Automatically by Keywords	Filtered Manually for Cloud Experiments	Citations for selected 44 articles
1,867	138	44 (15 NSDI, 7 OSDI, 7 SOSP, 15 SC)	11,203

Finding 2.1 Cloud performance variability is largely disregarded when researchers evaluate and prototype distributed systems, or compare established systems.

F2.2 Most cloud performance studies are under-specified. Most studies: (i) do not specify which performance measures are reported (i.e., mean, median); (ii) do not report minimal statistical variation data (i.e., standard deviation, quartiles); (iii) do not report the number of repetitions of an experiment.

F2.3 Most cloud performance evaluations are poorly designed: a large majority of such studies only perform small numbers of experiment repetitions (i.e., 3-10 trials), and do not assess variability or confidence.

Over the last decade, big data platforms and applications have been co-evolving with the cloud. This allowed researchers and practitioners to develop, deploy, and evaluate their applications and systems on various virtualized infrastructures. There is much evidence that clouds suffer from performance variability [7, 13, 35, 47]. It is therefore intuitive to ask if practitioners and system designers take variability into account when designing experiments or building systems. To answer these questions, we performed a systematic literature survey covering prominent conferences in the field: NSDI [44], OSDI [5], SOSP [1], and SC [2].

Survey Methodology: Table 1 shows the parameters of our survey, and Table 2 presents our survey process in-depth: (1) we started with all articles published in the aforementioned venues; (2) selected automatically a subset, based on string matching our query on keywords, title, and abstract; (3) we manually selected the articles in which the experiments were performed on a public cloud. The 44 selected articles are highly influential, having been cited **11,203** times so far[1].

The criteria we looked for when analyzing such articles

[1]according to Google Scholar on May 20, 2019

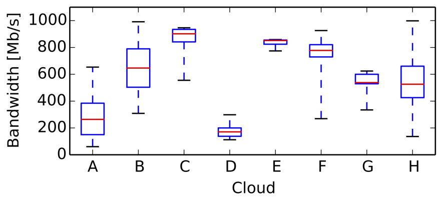


Figure 2: Bandwidth distributions for eight real-world clouds. Box-and-whiskers plots show the 1st, 25th, 50th, 75th, and 99th percentiles. (Distributions derived from the study [7] conducted by Ballani et al.)

are the following: (i) reporting average or median metrics over a number of experiments; (ii) reporting variability (such as standard deviation or percentiles) or confidence (such as confidence intervals); (iii) reporting the number of times an experiment was repeated. These are all critical criteria for determining whether a study's conclusions may be irreproducible, or worse, not fully supported by the evidence (i.e., flawed). To check the reliability of our manual filtering, it was performed by two separate reviewers, and we applied Cohen's Kappa coefficient [16] for each category presented in Figure 1a: reporting average or median, statistics, and poor specification. Our Kappa scores for each category, were 0.95, 0.81, and 0.85, respectively. Values larger than 0.8 are interpreted as near-perfect agreement between the reviewers [61].

Survey Results: *The systems community centered around cloud computing and big data disregards performance variability when performing empirical evaluations in the cloud.* Figure 1 shows the results of our survey. Out of the two reviewer's scores, we plot the lower scores, i.e., ones that are more favorable to the articles. We found that over 60% of the surveyed articles are under-specified (i.e., the authors do not mention how many times they repeated the experiments or even whether they are reporting average, median, etc.); a subset of the articles report averages or medians, but out of those, only 37% report variance or confidence (i.e., error-bars, percentiles). We further found that most articles that do report repetitions perform only 3, 5 or 10 repetitions of the experiments. The reason for such practices might be that experimenters are more used to evaluating software in controlled environments—what is true in controlled environments often does not hold in clouds.

Moreover, 76% of the properly specified studies use no more than 15 repetitions. Coupled with the effects of cloud variability, such experiment design practices could lead to wrong or ambiguous conclusions, as we show next.

2.1 How credible are experiments with few repetitions?

Experiments with few repetitions run the risk of reporting inaccurate results; the higher the variability, the greater the risk that a low-repetition experiment's results are unreliable. We

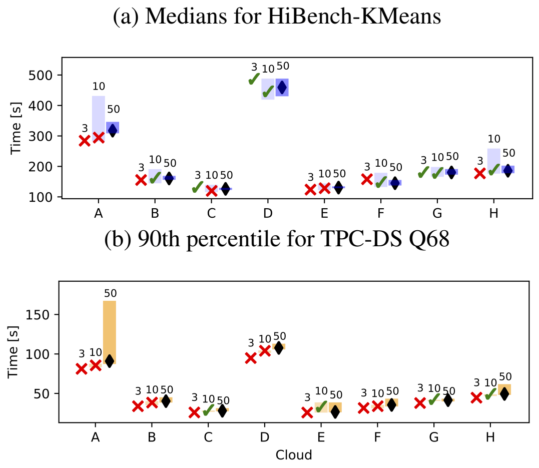


Figure 3: Medians and 90th percentiles for K-Means (a) and TPC-DS Q68 (b). Estimates are shown along with their 95% confidence intervals (CIs) for performance measurements under the A-H distributions. $\diamond$ depicts estimates 50-runs. Judged by the 50-run CIs we consider *gold standard*, accurate estimates (inside those CIs) are ✓; inaccurate estimates (outside those CIs) are $\times$ for 3- and 10-run sets.

use application-level benchmarks to show how the bandwidth distributions found by Ballani et al. [7] for eight real-world clouds—shown in Figure 2—do affect findings in practice.

We emulate the behavior of the eight clouds presented in Figure 2, which were contemporary with most articles found in our survey. In a private Spark [4] cluster of 16 machines, we limit the bandwidth achieved by machines according to distributions $A-H$. We uniformly sample bandwidth values from these distributions every $x \in \{5,50\}$ seconds. We used 50 experiment repetitions as our "gold standard" to demonstrate the intuition that running more experiments yields more accurate results, and compared them to the 3- and 10-repetitions commonly found in our literature survey. (In Section 4 we propose better methods for experiment design.)

Emulation Results: We found that experiments with few repetitions often produced medians that are *outside of* the 95% confidence intervals (CIs) for larger experiment sequences. The 95% CIs for medians represent ranges in which we would find true medians with 95% probability, if we were able to run infinite repetitions. Thus, when the low-repetition medians lie outside of the high-repetition CIs, there is a 95% probability that the former are inaccurate. This can be seen in Figure 3, which plots estimates of 95% nonparametric (asymmetric) CIs [11] for experiments using bandwidth distributions $A-H$ from Figure 2. For each bandwidth distribution, we show the medians and CIs for 3-, 5-, and 50-repetition experiments.[2] The median for the "gold standard" experiment is marked with a diamond; medians for lower-repetition experiments are shown with an "X" if outside the gold-standard 95% CI, or a check-mark if within it.

[2]Three repetitions are insufficient to calculate CIs; we plot medians because this is representative of what is often found in the literature.

The top of Figure 3 (part (a)) shows our estimates of *medians* for the K-Means application from HiBench [32]. Of the eight cloud bandwidth distributions, the 3-run median falls outside of the gold-standard CI for six of them (75%), and the 10-run median for three (38%). The bottom half of Figure 3 (part (b)) shows the same type of analysis, but this time, for tail performance [22] instead of the median. To obtain these results, we used TPC-DS [49] Query-68 measurements and the method from Le Boudec [11] to calculate nonparametric estimates for the 90th percentile performance, as well as their confidence bounds. As can be seen in this figure, it is even more difficult to get robust *tail* performance estimates.

Emulation Methodology: The quartiles in Ballani's study (Figure 2) give us only a rough idea about the probability densities and there is uncertainty about fluctuations, as there is no data about sample-to-sample variability. Considering that the referenced studies reveal no autocovariance information, we are left with using the available information to sample bandwidth uniformly. Regarding the sampling rate, we found the following: (1) As shown in Section 3 two out of the three clouds we measured exhibits significant sample-to-sample variability on the order of tens of seconds; (2) The cases F-G from Ballani's study support fine sampling rates: variability at sub-second scales [63] and at the 20s intervals [24] is significant. Therefore, we sample at relatively fine-grained intervals: 5s for Figure 3(a), and 50s for Figure 3(b). Furthermore, sampling at these two different rates shows that benchmark volatility is not dependent on the sampling rate, but rather on the distribution itself.

3 How Variable Are Cloud Networks?

We now gather and analyze network variability data for three different clouds: two large-scale commercial clouds, and a smaller-scale private research cloud. Our main findings can be summarized as follows:

F3.1 Commercial clouds implement various mechanisms and policies for network performance QoS enforcement, and these policies are opaque to users and vary over time. We found (i) token-bucket approaches, where bandwidth is cut by an order of magnitude after several minutes of transfer; (ii) a per-core bandwidth QoS, prioritizing heavy flows; (iii) instance types that, when created repeatedly, are given different bandwidth policies unpredictably.

F3.2 Private clouds can exhibit more variability than public commercial clouds. Such systems are orders of magnitude smaller than public clouds (in both resources and clients), meaning that when competing traffic does occur, there is less statistical multiplexing to "smooth out" variation.

F3.3 Base latency levels can vary by a factor of almost 10x between clouds, and implementation choices in the cloud's virtual network layer can cause latency variations over two orders of magnitude depending on the details of the application.

Table 3: Experiment summary for determining performance variability in modern cloud networks. Experiments marked with a star (*) are presented in depth in this article. Due to space limitations, we release the other data in our repository [59]. All Amazon EC2 instance types are typical offerings of a big data processing company [20].

Cloud	Instance Type	QoS (Gbps)	Exp. Duration	Exhibits Variability	Cost ($)
*Amazon	c5.XL	≤ 10	3 weeks	Yes	171
Amazon	m5.XL	≤ 10	3 weeks	Yes	193
Amazon	c5.9XL	10	1 day	Yes	73
Amazon	m4.16XL	20	1 day	Yes	153
Google	1 core	2	3 weeks	Yes	34
Google	2 core	4	3 weeks	Yes	67
Google	4 core	8	3 weeks	Yes	135
*Google	8 core	16	3 weeks	Yes	269
HPCCloud	2 core	N/A	1 week	Yes	N/A
HPCCloud	4 core	N/A	1 week	Yes	N/A
*HPCCloud	8 core	N/A	1 week	Yes	N/A

3.1 Bandwidth

We run our bandwidth measurements in two prominent commercial clouds, Amazon EC2 (us-east region) and Google Cloud (us-east region), and one private research cloud, HPCCloud[3]. Table 3 summarizes our experiments. In the interest of space, in this paper we focus on three experiments; all data we collected is available in our repository [59]. We collected the data between October 2018 and February 2019. In total, we have over 21 weeks of nearly-continuous data transfers, which amount for over 1 million datapoints and over 9 petabytes of transferred data.

The Amazon instances we chose are typical instance types that a cloud-based big data company offers to its customers [20], and these instances have AWS's "enhanced networking capabilities" [6]. On Google Cloud (GCE), we chose the instance types that were as close as possible (though not identical) to the Amazon EC2 offerings. HPCCloud offered a more limited set of instance types. We limit our study to this set of cloud resources and their network offerings, as big data frameworks are not equipped to make use of more advanced networking features (i.e., InfiniBand), as they are generally designed for commodity hardware. Moreover, vanilla Spark deployments, using typical data formats such as Parquet or Avro, are not able to routinely exploit links faster than 10 Gbps, unless significant optimization is performed [58]. Therefore, the results we present in this article are highly likely to occur in real-world scenarios.

In the studied clouds, for each pair of VMs of similar instance types, we measured bandwidth continuously for one week. Since big data workloads have different network access patterns, we tested multiple scenarios:

- **full-speed** - continuously transferring data, and summarizing performability metrics (bandwidth, retransmis-

[3]https://userinfo.surfsara.nl/systems/hpc-cloud

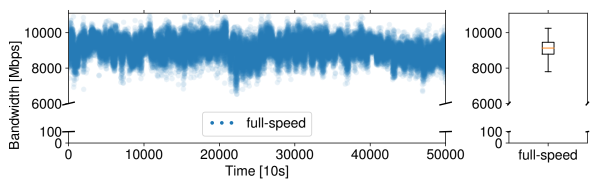


Figure 4: Variable network bandwidth performance in the HPCCloud (left); the statistical performance distribution, plotted as an IQR box; the whiskers represent 1st and 99th percentiles (right). Duration: a week of continuous experimentation; each point is average over 10 seconds.

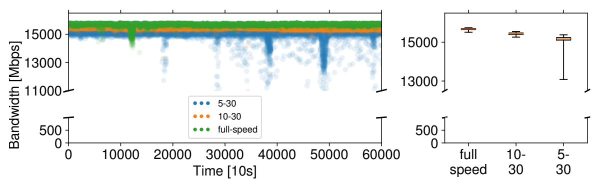


Figure 5: Variable network bandwidth performance in the Google Cloud (left), and the statistical performance distribution, plotted as an IQR box, where the whiskers are 1st and 99th percentiles (right). The duration is a week of continuous experimentation, each point is an average over 10 seconds.

sions, CPU load etc.) every 10 seconds;

- **10-30** - transfer data 10 seconds, wait 30 seconds;
- **5-30** - transfer data 5 seconds, wait 30 seconds.

The first transmission regime models highly network intensive applications, such as long-running batch processing or streaming. The last two modes mimic short-lived analytics queries, such as TPC-H, or TPC-DS.

HPCCloud. Small-scale (i.e., up to 100 physical machines and several hundred users) private (research) clouds often do not use mechanisms to enforce network QoS. We measured the network performance variability between pairs of VMs, each having 8 cores. Figure 4 plots the results. We show our measurements only for "full-speed" (i.e., continuous communication) because our other experiments show similar behavior. We observe that the network bandwidth shows high variability, ranging from 7.7 Gbps to 10.4 Gbps.

Google Cloud. GCE states that it enforces network bandwidth QoS by guaranteeing a “per-core” amount of bandwidth. Our measurements, plotted in Figure 5, fall close to the QoS reported by the provider, but access pattern affects variability to a greater degree than in other clouds. Longer streams (*full-speed*) exhibit low variability and better overall performance, while *5-30* has a long tail. This could be due to the design of the Google Cloud network, where idle flows use dedicated gateways for routing through the virtual network [18]. We observe that network bandwidth varies significantly, depending on access patterns, between 13 Gbps and 15.8 Gbps.

Amazon EC2. We discover the opposite behavior in EC2: heavier streams achieve less performance and more variability compared to lighter (shorter) streams, as shown in Figure 6. Considering the large performance differences between

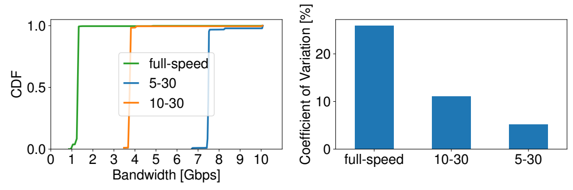


Figure 6: Variable network bandwidth performance in Amazon EC2, plotted as an empirical cumulative distribution (left), barplot of the coefficient of variation (right). The duration is a week of continuous experimentation, each data point representing an average over 10 seconds.

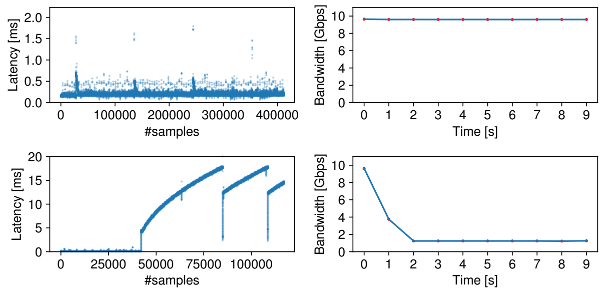


Figure 7: Example of observed Amazon EC2 latency for a 10-second TCP sample on *c5.xlarge*. Left: RTT latency for TCP packets. Right: achieved *iperf* bandwidth. Top: regular Amazon EC2 behavior. Bottom: latency behavior when a drop in bandwidth occurs.

these experiments, we plot our measurements as a CDF and a barplot of coefficient of variation to improve visibility. There are approximately 3x and 7x slowdowns between *10-30* and *5-30* and *full-speed*, respectively. The achieved bandwidth varies between 1 Gbps and 10 Gbps. We investigate the causes of this behavior in Section 3.3.

How rapidly does bandwidth vary? Our analysis shows the level of *measurement-to-measurement* variability is significant: bandwidth in HPCCloud (*full-speed*) and Google Cloud (*5-30*) varies between consecutive 10-second measurements up to 33% and 114%, respectively. While a small sample may exhibit only modest fluctuations, the long-tailed distributions we observed here strongly suggest using the analysis techniques we discuss in Section 4.1. Amazon EC2’s variability is more particular and policy-dependent (Section 3.3).

3.2 Latency

Commercial clouds implement their virtual networks using very different mechanisms and policies. We can see this in more detail by looking at the round-trip lantencies seen in Google Cloud and Amazon EC2. We measure the application-observed TCP RTT, as this is what impacts the high-level networking stacks of big data frameworks. For our experiments, we run 10-second streams of *iperf* tests, capturing all packet headers with *tcpdump*. We perform an offline analysis of the packet dumps using *wireshark*, which compares the

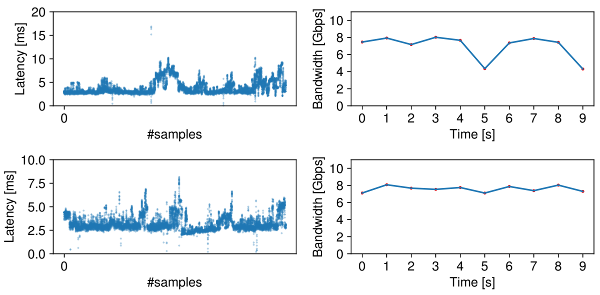


Figure 8: Example of observed Google Cloud latency for a 10-second TCP sample on a *4-core* instance. Left: RTT latency for TCP packets. Right: achieved *iperf* bandwidth.

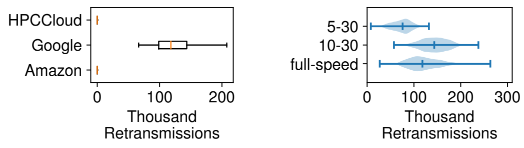
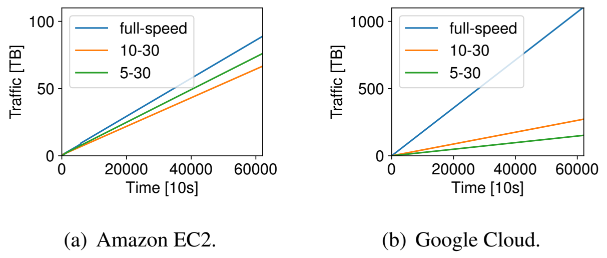


Figure 9: TCP retransmission analysis, summarized for all experiments presented before, in all clouds. Left: retransmissions as IQR boxplots, with the whiskers representing 1st and 99th percentiles; Right: violin plot for retransmissions in Google Cloud; thickness of the plot is proportional to the probability density of the data.

time between when a TCP segment is sent to the (virtual) network device and when it is acknowledged. Our data was collected between August and September 2019. In total, it contains over 50 million RTT datapoints.

The behavior we observe is inherently different: Google Cloud exhibits latency in the order of milliseconds, with an upper limit of 10ms. Amazon EC2 generally exhibits faster sub-millisecond latency under typical conditions, but when the traffic shaping mechanism (detailed in Section 3.3) takes effect, the latency increases by two orders of magnitude, suggesting large queues in the virtual device driver. Figure 7 shows representative patterns of latency in the Amazon EC2 cloud, while Figure 8 is representative of Google Cloud. Both figures plot latency as RTT packet data obtained from a 10-second TCP stream obtained running an *iperf* benchmark.

The behavior observed in the top half of Figure 7 lasts for approximately ten minutes of full-speed transfer on *c5.xlarge* instances. After this time, the VMs' bandwidth gets throttled down to about 1 Gbps (bottom half of Figure 7), which also significantly increases latency. On Google Cloud, there is no throttling effect, but the bandwidth and latency vary more from sample to sample.



(a) Amazon EC2. (b) Google Cloud.

Figure 10: The total amount of data transferred between the pairs of virtual machines involved in the three types of experiments performed. The total time is a week, while each point on the horizontal axis represents 10 seconds.

3.3 Identifying Mechanisms and Policies

The behavior exhibited by the two commercial providers is notably different. We uncover mechanisms and policies for enforcing client QoS by performing extra analysis, depicted in Figures 9 and 10. The former plots the number of retransmissions per experiment (part (a)) and a zoomed-in view of Google Cloud (part (b)). Amazon EC2 and HPCCloud have a negligible number of retransmissions, yet retransmission are common in Google Cloud: roughly 2% per experiment.

Figure 10 plots the total amount of traffic for Amazon EC2 and Google Cloud over the entire duration of our experiments. It is clear that in Google Cloud's case the amount of traffic generated by *full-speed* is orders of magnitude larger than for the intermittent access patterns. In Amazon EC2's case, the total amount of data sent for all three kinds of experiments is roughly equal. By corroborating this finding the more fine-grained experiments we performed presented in Figure 7, and other empirical studies [51, 62], we find that Amazon EC2 uses a *token-bucket* algorithm to allocate bandwidth to users.

Token-Bucket Analysis. The token-bucket algorithm operation can be explained as follows. When a VM is provided to the user, its associated *bucket* holds a certain amount of tokens (i.e., a budget). This budget is allowed to be spent at a high rate (i.e., 10 Gbps). When the budget is depleted (e.g., after about 10 minutes of continuous transfer on a *c5.xlarge* instance, the QoS is limited to a low rate (e.g., 1 Gbps). The bucket is also subject to a replenishing rate that we empirically found to be approximately 1 Gbit token per second, i.e., every second users receive the amount of tokens needed to send 1 Gbit of data at the high (10 Gbps) rate. Once the token bucket empties, transmission at the capped rate is sufficient to keep it from filling back up. The user must *rest* the network, and re-filling the bucket completely takes several minutes.

We analyze the behavior of multiple types of VMs from the *c5.** family, and find that their token-bucket parameters differ. More expensive machines benefit from larger initial budgets, as well as higher bandwidths when their budget depletes. Figure 11 plots the token-bucket parameter analysis for four VMs of the *c5.** family. For each VM type, we ran an *iperf* test continuously until the achieved bandwidth dropped significantly and stabilized at a lower value. For each instance type, we ran 15 tests. Figure 11 shows the time taken to empty the token bucket, the *high* (non-empty bucket) bandwidth value, and the *low* (empty bucket) bandwidth value. As the *size* (i.e., number

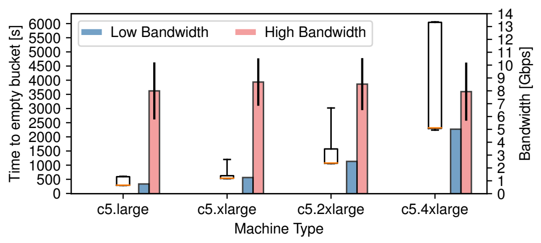


Figure 11: The token-bucket parameters identified for several instances of Amazon EC2 *c5.** family. The elapsed time to empty the token bucket is depicted with boxplots associated with *left* vertical axis. The *high* and *low* bandwidths of the token bucket are depicted with bar plots with whiskers and are associated with the *right* vertical axis.

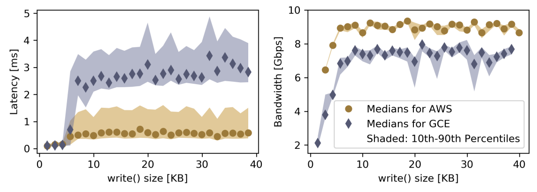


Figure 12: Measured latency and bandwidth for Amazon EC2 (*c5.xlarge*) and GCE (4-core VM with advertised 8 *Gbps*) instances as functions of the `write()` size.

of cores, amount of memory etc,) of the VM increases, we notice that the *bucket size* and the *low* bandwidth increase proportionally. However, as the magnitude of the boxplots suggests, as well as the error bars we plotted for the *high* bandwidth, these parameters are not always consistent for multiple incarnations of the same instance type.

Virtual NIC Implementations. We found that differences in EC2 and GCE's implementations of virtual NICs can lead to significantly different observed behavior. EC2's virtual NICs advertise an MTU of 9000 bytes, a standard "jumbo frame" size. GCE's only advertise an MTU of 1500 bytes (standard Ethernet frame size), but instead enable TCP Segmentation Offloading (TSO), in which the NIC accepts larger "packets" from the device driver, but then breaks them down into smaller Ethernet frames before transmission (we do not know whether this occurs at the virtual or physical NIC in GCE's implementation). Both of these techniques serve the same basic function—reducing overhead by sending fewer, larger packets on the virtual NIC, but result in different observable behavior on the host, and the details of this behavior depend heavily on the application and workload.

The most striking effect is the way that the size of the `write()`s done by the application affects latency and packet retransmission. Figure 12 plots the effects of the `write()` size on latency and bandwidth. On EC2, the size of a single "packet" tops out at the MTU of 9K, whereas on GCE, TSO can result in single "packet" at the virtual NIC being as large as 64K in our experiments. With such large "packets," perceived latency increases greatly due to the higher perceived "transmission time" for these large packets. The number of retransmissions also goes up greatly, presumably due to limited buffer space in the bottom half of the virtual NIC driver or tighter bursts on the physical NIC. In practice, the size of the "packets" passed to the virtual NIC in Linux tends to equal to the `write` on the socket (up to the cap noted above). This makes the observed behavior—and thus repeatability and the ability to generalize results between clouds—highly application-dependent. It is also worth noting that all streams are affected when one stream sends large "packets", since they share a queue in the virtual device driver. On GCE, when we limited our benchmarks to `write`s of 9K, we got near-zero packet retransmission and an average RTT of about 2.3*ms*. When the benchmark used its default `write()` size of 128K, we saw the hundreds of thousands of retransmission shown in Figure 9 and latencies as high as 10*ms*.

4 Performance Reproducibility For Big Data Applications

Having looked at low-level variability in bandwidth and latency, we now move "up" a level to applications and workloads. Our main findings are:

F4.1 Under variability resembling Google Cloud and HPCCloud, which can be modeled as stochastic noise, reproducible experiments can be obtained using sufficient repetitions and sound statistical analyses.

F4.2 Application transfer patterns exhibit non-trivial interactions with token-bucket network traffic shapers. Depending on the bucket budget and the application, significant application performance variability is incurred.

F4.3 Token-bucket traffic shapers in conjunction with (imbalanced) big data applications can create stragglers.

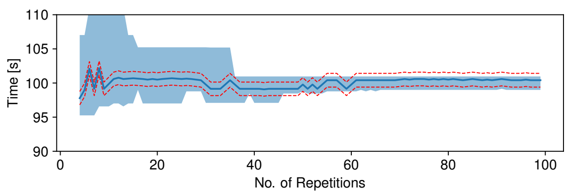
F4.4 In long-running cloud deployments that have incurred large amounts of varied network traffic, it is highly difficult to predict application performance, as it is dependent on the state of the individual nodes' remaining token-bucket budgets.

Big Data Workloads. In this section, we run the HiBench [32] and TPC-DS [49] benchmarks on Spark 2.4.0 (see Table 4) to showcase our main findings on network variability and big data workloads reproducibility. In 2015, Ousterhout et al. [50] found that big data workloads are mostly CPU bound. The workloads we chose here are no exception. However, they are *sensitive* to oscillations in the network transfer performance. Moreover, most of the CPU load in [50] is attributed to the framework's inefficiencies [19], which have been solved in later releases. As a consequence, modern Spark implementations are more sensitive to network variations.

4.1 Experiments and Stochastic Noise

As detailed in Section 3, the behavior of network performance variability for Google Cloud and HPCCloud is closer in na-

(a) Median Performance for K-Means in Google Cloud.



(b) Median Performance for TPC-DS Q65 in HPCCloud.

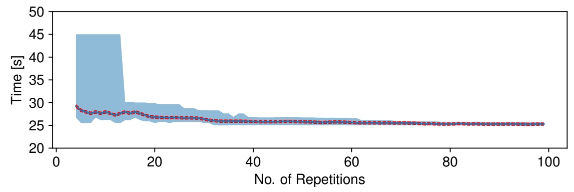


Figure 13: CONFIRM analysis for K-Means and TPC-DS Q65 on Google Cloud and HPCCloud. Median estimates (blue thick curve), 95% nonparametric confidence intervals (light blue filled space), and 1% error bounds (red dotted curves). Vertical axis not starting at 0 for visibility.

Table 4: Big data experiments on modern cloud networks.

Workload	Size	Network	Software	#Nodes
HiBench [32]	BigData	Token-bucket, Figure 14	Spark 2.4.0, Hadoop 2.7.3	12
TPC-DS [49]	SF-2000	Token-bucket, Figure 14	Spark 2.4.0, Hadoop 2.7.3	12

ture to stochastic variability given by transient conditions in the underlying resources, such as noisy neighbors. To achieve reproducible experiments under such conditions, system designers and experimenters need to carefully craft and plan their tests, using multiple repetitions, and must perform sound statistical analyses.

We ran several HiBench [32] and TPC-DS [49] benchmarks directly on the Google Cloud and HPCCloud clouds and report how many repetitions an experimenter needs to perform in order to achieve trustworthy experiments. While it is true that running experiments directly on these clouds we cannot differentiate the effects of network variability from other sources of variability, the main take-away message of this type of experiment is that this kind of stochastic variability can be accounted for with proper experimentation techniques.

On the performance data we obtained, we performed a CONFIRM [47] analysis to predict how many repetitions an experiment will require to achieve a desired confidence interval. Figure 13 presents our findings, showing that for these two common benchmarks, it can take 70 repetitions or more to achieve 95% confidence intervals within 1% of the measured median. As we saw in Section 2, this is far more repetitions than are commonly found in the literature: most papers are on the extreme left side of this figure, where the confidence intervals are quite wide. This points to the need for stronger experiment design and analysis in our community.

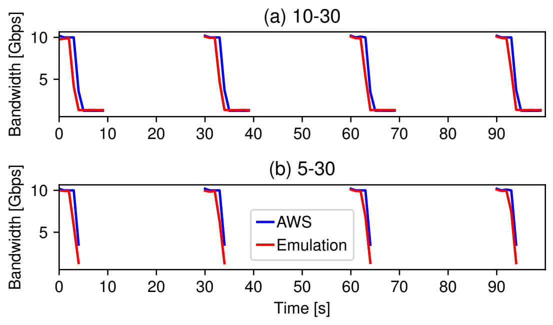


Figure 14: Validation of the emulation of the token-bucket policy of Amazon EC2. The similar aspect of the two curves indicates that emulation is high-quality.

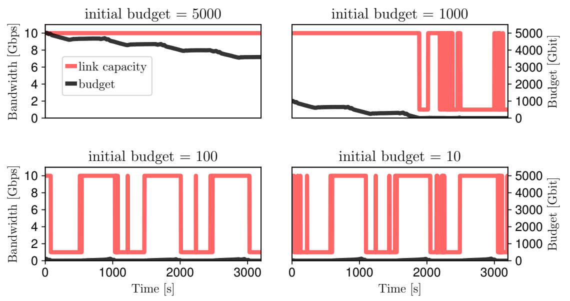


Figure 15: Link capacity allocated when running Terasort on a token bucket. Left vertical axis shows the link capacity; right vertical axis shows the token bucket budget. Budget depletes due to application network transfers.

4.2 Experiments and Token-Buckets

In contrast to Google Cloud and HPCCloud, the *token-bucket* shaping policy of Amazon EC2 is *not* stochastic noise, and needs in-depth analysis. Because token-bucket behavior is dependent on past network access patterns, *an application influences not only its own runtime, but also future applications' runtimes*.

Token-bucket Emulator. We decided to emulate the behavior of Amazon EC2 token-bucket instead of directly running applications in this cloud. We believe this type of experimentation is superior to the other two alternatives: (i) simulation, or (ii) directly running applications on the cloud. For the former, we believe the behavior of big data applications under network performance variability is far too subtle and complex to properly simulate while modeling and capturing all possible variables. For the latter, we perform the emulation in an isolated setup, i.e., a private cluster, that does not share resources. This allows us to test in isolation the effects of network performance variability, excluding as much as possible all other sources of variability one could encounter in a cloud (e.g., CPU, memory bandwidth, I/O etc.). If we were to directly run applications in a cloud, it would have been difficult to separate the effects of network variability

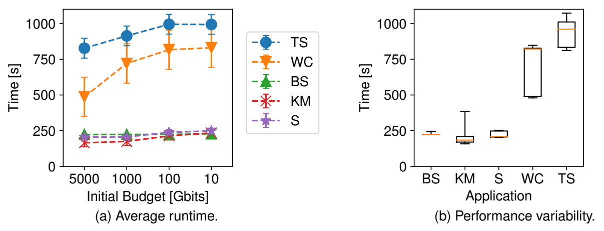


Figure 16: HiBench average runtime (left) and performance variability (right), plotted as IQR box (whiskers represent 1st and 99th percentiles), induced by token bucket budget variability. The more network-dependent applications are affected more by lower budgets.

from, for example, the effects of CPU variability.

We built a network emulator based on the Linux `tc` [33] facility. Figure 14 plots the real-world behavior encountered in Amazon EC2 in comparison with our emulation. This experiment is a zoomed-in view of the experiment in Section 3.1, where our servers were communicating for either five or ten seconds, then slept for 30 seconds. At the beginning of each experiment, we made sure that the token-bucket budget is nearly empty. During the first few seconds of the experiment the token-bucket budget gets completely exhausted. For each sending phase of 5 or 10 seconds, the system starts at a high QoS (10 Gbps bandwidth), after a few seconds the budget is emptied, and the system drops to a low QoS (1 Gbps).

Experiment Setup. We perform the experiments described in Table 4 on a 12-node cluster. Each node has 16 cores, 64GB memory, a 256GB SSD, and FDR InfiniBand network. Using the emulator presented in Figure 14, we run on the emulated Amazon EC2 token-bucket policy all applications and queries in the HiBench [32] and TPC-DS [49] benchmark suites. The emulated setup is that of the *c5.xlarge* instance type, which typically sees a high bandwidth of 10 Gbps and a low bandwidth of 1 Gbps. Throughout our experiments we vary the token bucket budget to assess its impact on big data applications. We run each workload a minimum of 10 times for each token-bucket configuration and report full statistical distributions of our experiments.

Token-bucket-induced Performance Variability. One important parameter for the token-bucket is its budget: the number of tokens available at a certain moment in time. This is highly dependent on the previous state of the virtual machine (i.e., how much network traffic has it sent recently), and has a large impact on the performance of future deployed applications. Note that it is difficult to estimate the currently-available budget for anything other than a "fresh" set of VMs: each VM has its own token bucket, the remaining budget is a function of previous runs, and, as we saw in Figure 11 the constants controlling the bucket are not always identical.

Application performance is highly dependent on the budget, and deployments with smaller budgets create more network performance variability. Figure 15 shows the network traffic behavior of the Terasort application with different initial budgets. For each budget, the subfigures show the application network profile for 5 consecutive runs. We notice a strong correlation between small budgets and network performance variability: there is much more variability for budgets $\in \{10,100\}$ Gbits, than for budgets $\in \{1000,5000\}$ Gbits.

Figure 16 shows how this effect manifests in the runtimes of HiBench: it plots the average application runtime (left) over 10 runs for budgets $\in \{10,100,1000,5000\}$ Gbits, and the performance variability over the same budgets (right). For the more network-intensive applications (i.e., TS, WC), the initial state of the budget can have a 25%–50% impact on performance.

A similar behavior is observed for the TPC-DS benchmark suite. Figure 17 shows the query sensitivity to the token budget and the variability induced by different budget levels. Figure 17(a) plots average runtime slowdown for 10-run sets of TPC-DS queries for budgets $\in \{10,100,1000\}$ Gbits, compared to the 5000 Gbit budget. For all queries, larger budgets lead to better performance. Figure 17(b) plots the performance variability over all tested budgets. Queries with higher network demands exhibit more sensitivity to the budget and hence higher performance variability.

These results clearly show that if the system is left in an unknown state (e.g., a partially-full token bucket, left over from previous experiments), the result is likely to be an inaccurate performance estimate. Evidence from Figures 16(b) and 17(b) strongly supports this, as performance varies widely for the network-intensive queries and applications depending on the token-bucket budget.

Token-bucket-induced Stragglers. Non-trivial combinations of token-bucket budgets, application scheduling imbalances, and network access patterns lead to straggler nodes. Figure 18 shows that for budget = 2500 Gbits and application TPC-DS, the application gets slowed down by a straggler: all nodes but one in the deployment do not deplete their budgets completely, thus remaining at a high bandwidth QoS of 10 Gbps. However, there is one node on which the token-bucket budget is depleted, causing its bandwidth to get limited to 1 Gbps. Exacerbating the variability, the behavior is not consistent: this node oscillates between high and low bandwidths in short periods of time. Such unpredictable behavior leads to both performance variability of the entire setup and also poor experiment reproducibility. This behavior will be prevalent in many *unbalanced* networked applications, where certain servers might perform more transfers than others. Especially in long-running clusters, the state of the individual servers' token-buckets will be highly different. As a direct consequence, the overall system will suffer from stragglers.

Repeatable experiments and token-buckets. Token-bucket policies for enforcing network QoS can have unexpected and detrimental impacts on sound cloud-based experimentation. To explore this, we compute medians and their nonparametric confidence intervals (CIs), similar to the work

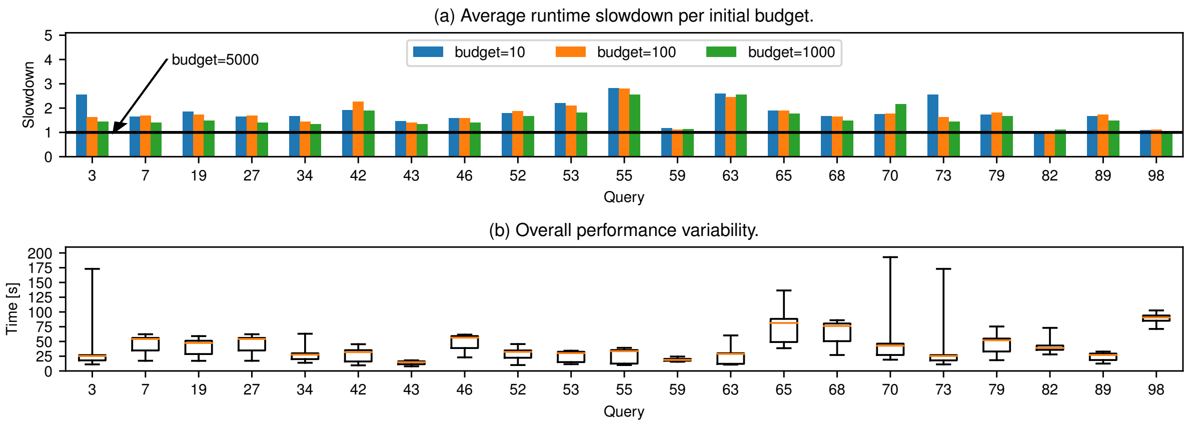


Figure 17: TPC-DS average runtime slowdown per query depending on initial budget (top); overall performance variability, summarized over initial budgets (bottom), plotted as IQR box; whiskers represent 1st and 99th percentiles.

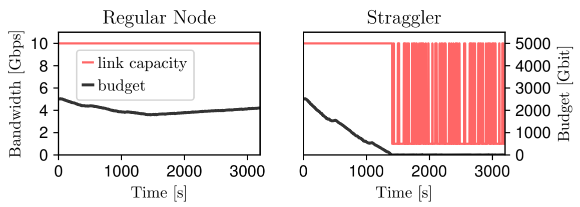


Figure 18: Link capacity allocation for TPC-DS on a token-bucket network, with initial budget = 2500 Gbit. Regular node network utilization (left); straggler node (right).

by Maricq et al. [47], across a number of initial token budgets. Figure 19 plots median estimates for two TPC-DS queries, along with 95% CIs and 10% error bounds around medians. Repetitions of the experiments are independent: each one runs on fresh machines with flushed caches, and at the the beginning of each repetition, we reset the token budget. We reduce this initial budget over time to emulate the effects that previous experiments can have on subsequent ones: what this models is an environment in which many different experiments (or repetitions of the same experiment) are run in quick succession. This is likely to happen when running many experiments back-to-back in the same VM instances.

Query 82 (in the top of Figure 19) is agnostic to the token budget. Running more repetitions of this experiment tightens the confidence intervals, as is expected in CI analysis. In contrast, query 65 (in the bottom of the figure) depends heavily on the bucket budget; as a result, as we run more experiments, depleting the bucket budget, the query slows down significantly, and the initial CI estimates turn out to be inaccurate. In fact, the CIs *widen* with more repetitions, which is unexpected for this type of analysis. This is because the token bucket breaks the assumption that experiments are independent: in this model, more repetitions deplete the bucket that the next experiment begins with. These two queries represent extremes, but, as shown in the bar graph at the bottom of the figure, 80% of all queries we ran from TPC-DS suffer effects like Query 65: most produce median estimates that are more than 10% incorrect by the time we fully deplete the budget.

This demonstrates that, when designing experiments, we cannot simply rely on the intuition that more repetitions lead to more accurate results: we must ensure that factors hidden in the cloud infrastructure are reset to known conditions so that each run is truly independent. Others have shown that cloud providers use token buckets for other resources such as CPU scheduling [62]. This affects cloud-based experimentation, as the state of these token buckets is not directly visible to users, nor are their budgets or refill policies.

5 Summary: Is Big Data Performance Reproducible in Modern Cloud Networks?

We return to our two basic questions: (1) *How reproducible are big data experiments in the cloud?*; and (2) *What can experimenters do to make make sure their experiments are meaningful and robust?* Our findings are:

F5.1: Network-heavy experiments run on different clouds cannot be directly compared. Building a cloud involves trade-offs and implementation decisions, especially at the virtualization layer. Some of these decisions are well-documented by the platforms [6,28], but others, including the ones we have examined in this paper, are not. Unfortunately, these differences can cause behaviors that result in different application performance, such as the bandwidth differences seen in Figure 10 or the latency effects seen in Figure 12.

Both of these effects are rather large, and are dependent on factors such as the size of the application's write buffer and specific patterns of communication. While these decisions presumably serve the clouds' commercial customers well, they complicate things for those who are trying to draw scientific conclusions; when comparing to previously-published performance numbers, it is important to use the same cloud to ensure that differences measured are that of the systems

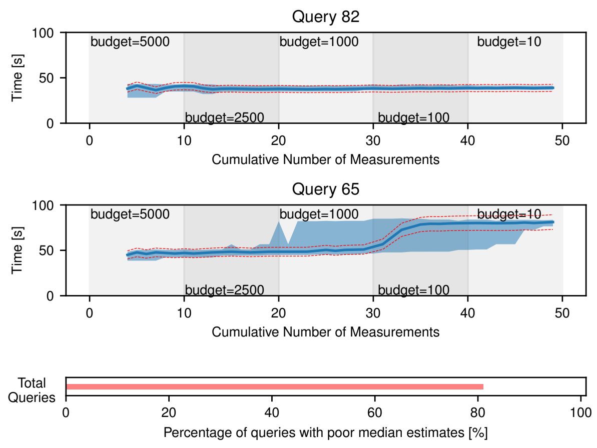


Figure 19: Median estimates (blue thick curve), 95% non-parametric confidence intervals (light blue filled space), and 10% error bounds (red dotted curves) for running two TPC-DS queries, over 5 token-bucket budgets. Bottom: number of queries for which we cannot achieve tight confidence intervals and accurate median estimates.

under test, and not artifacts of the cloud platform. Running on multiple clouds, can, however, be a good way to perform sensitivity analysis [36]: by running the same system with the same input data and same parameters on multiple clouds, experimenters can reveal how sensitive the results are to the choices made by each provider.

F5.2: Even within a single cloud, it is important to establish baselines for expected network behavior. These baselines should be published along with results, and need to be verified before beginning new experiments. Because cloud providers' policies can be opaque, and implementation details can change over time, it is possible for changes to invalidate over time experiments within the same cloud. For example, after several months of running experiments in Amazon EC2, we began encountering new behavior: prior to August 2019, all `c5.xlarge` instances we allocated were given virtual NICs that could transmit at 10 Gbps. Starting in August, we started getting virtual NICs that were capped to 5 Gbps, though not consistently (this behavior is part of the underlying cause of the distributions in Figure 11). The reasons for this are not clear, and we have no way to know whether the "new" behavior is a transient effect in response to increased congestion that month or a new, permanent policy.

If one can establish baseline expectations for how the platform will perform, and incorporate checks for them into the experimental process [37], one can at least detect when changes have occurred. Experimenters should check, through micro-benchmarks, whether specific cloud resources (e.g., CPU, network) are subject to provider QoS policies.

As opposed to contention-related variability, this type of variability is deterministic under carefully selected micro-benchmarks. In the network, these microbenchmarks should at a minimum include base latency, base bandwidth, how latency changes with foreground traffic, and the parameters to bandwidth token-buckets, if they are present. Furthermore, when reporting experiments, always include these *performance fingerprints* together with the actual data, as possible changes in results in the future could be explained by analyzing the micro-benchmark logs.

F5.3: Some cloud network variability (in particular, interference from neighbors) can be modeled as stochastic noise, and classic techniques from statistics and experiment design are sufficient for producing robust results; however, this often takes more repetitions than are typically found in the literature. Standard statistical tools such as ANOVA and confidence intervals [11, 36, 47] are effective ways of achieving robust results in the face of random variations, such as those caused by transient "noisy neighbors"; however, in order to be effective, they require many repetitions of an experiment, and, as we saw in Section 2, this bar is often not met in the literature. The more variance, the more repetitions are required, and as we saw in Figures 4, 5, and 6, network variance in the cloud can be rather high, even under 'ideal' conditions. An effective way to determine whether enough repetitions have been run is to calculate confidence intervals for the median and tail, and to test whether they fall within some acceptable error bound (e.g., 5% of value they are measuring).

F5.4: Other sources of variability cause behavior that breaks standard assumptions for statistical analysis, requiring more careful experiment design. Some of the variability we have seen (e.g., Figures 12, 18, and 19) causes behavior that breaks standard assumptions for statistical analysis (such as iid properties and stationarity). As an integral part of the experimentation procedure, samples collected should be tested for normality [56], independence [46], and stationarity [23]. When results are not normally-distributed, non-parametric statistics can be used [26]. When performance is not stationary, results can be limited to time periods when stationarity holds, or repetitions can be run over longer time frames, different diurnal or calendar cycles, etc. Techniques like CONFIRM [47] can be used to test whether confidence intervals converge as expected.

Discretizing performance evaluation into units of time, e.g., *one hour* is helpful. Gathering median performance for each interval, and applying techniques such as CONFIRM over large-numbers of gathered medians results in significant and realistic performance data. Large intervals can smooth out noise, helping to reduce unrepresentative measurements.

We also find it helpful to 'rest' the infrastructure and randomize [3] experiment order. Because it is hard to tell what performance-relevant state may build up in the hidden parts of the underlying cloud infrastructure, experimenters must ensure that the infrastructure is in as 'neutral' a state as possible at the beginning of every experiment. The most reliable way to do this is to create a fresh set of VMs for every experiment. When running many small experiments, this can be cost- or time-prohibitive: in these cases, adding delays between experiments run in the same VMs can help. Data used while gathering baseline runs can be used to determine the appropriate length (e.g., seconds or minutes) of these rests. Randomized experiment order is a useful technique for avoiding self-interference.

F5.5: Network performance on clouds is largely a function of provider implementation and policies, which can change at any time. Experimenters cannot treat "the cloud" as an opaque entity; results are significantly impacted by platform details that may or may not be public, and that are subject to change. (Indeed, much of the behavior that we document in Sections 3 and 4 is unlikely to be static over time.) Experimenters can safeguard against this by publishing as much detail as possible about experiment setup (e.g., instance type, region, date of experiment), establishing baseline performance numbers for the cloud itself, and only comparing results to future experiments when these baselines match.

Applicability to other domains. In this paper, we focused on big data applications and therefore our findings are most applicable in this domain. The cloud-network related findings we present in Section 3 are general, so practitioners from other domains (e.g., HPC) should take them in to account when designing systems and experiments. However, focusing in depth on other domains might reveal interactions between network variability and experiments that are not applicable to big data due to the intrinsic application characteristics. Therefore, while our findings in Section 4 apply to most other networked applications, they need not be complete. We also believe that a community-wide effort for gathering cloud variability data will help us automate reproducible experiment design that achieves robust and meaningful performance results.

6 Related Work

We have showed the extent of network performance variability in modern clouds, as well as how practitioners disregard cloud performance variability when designing and running experiments. Moreover, we have showed what the impact of network performance variability is on experiment design and on the performance of big data applications. We discuss our contributions in contrast to several categories of related work.

Sound Experimentation (in the Cloud). Several articles already discuss pitfalls of systems experiment design and presentation. Such work fits two categories: guidelines for better experiment design [3,17,38,47] and avoiding logical fallacies in reasoning and presentation of empirical results [10,21,31]. Adding to this type of work, we survey how practitioners apply such knowledge, and assess the impact of poor experiment design on the reliability of the achieved results. We investigate the impact of *variability* on performance reproducibility, and uncover variability behavior on modern clouds.

Network Variability and Guarantees. Network variability has been studied throughout the years in multiple contexts, such as HPC [8,9], experimental testbeds [47] and virtualized environments [35,40,55]. In the latter scenario, many studies have already assessed the performance variability of cloud datacenter networks [43, 51, 63]. To counteract this behavior, cloud providers tackle the variability problem at the infrastructure level [12,52]. In general, these approaches introduce network virtualization [30,54], or traffic shaping mechanisms [18], such as the token buckets we identified, at the networking layer (per VM or network device), as well as a scheduling (and placement) policy framework [41]. In this work, we considered both types of variability: the one given by resource sharing and the one introduced by the interaction between applications and cloud QoS policies.

Variability-aware Network Modeling, Simulation, and Emulation. Modeling variable networks [27,45] is a topic of interest. Kanev et al. [39] profiled and measured more than 20,000 Google machines to understand the impact of performance variability on commonly used workloads in clouds. Uta et al. emulate gigabit real-world cloud networks to study their impact on the performance of batch-processing applications [60]. Casale and Tribastone [14] model the exogenous variability of cloud workloads as continuous-time Markov chains. Such work cannot isolate the behavior of network-level variability compared to other types of resources.

7 Conclusion

We studied the impact of cloud network performance variability, characterizing its impact on big data experiment reproducibility. We found that many articles disregard network variability in the cloud and perform a limited number of repetitions, which poses a serious threat to the validity of conclusions drawn from such experiment designs. We uncovered and characterized the network variability of modern cloud networks and showed that network performance variability leads to variable slowdowns and poor performance predictability, resulting in non-reproducible performance evaluations. To counter such behavior, we proposed protocols to achieve reliable cloud-based experimentation. As future work, we hope to extend this analysis to application domains other than big data and develop software tools to automate the design of reproducible experiments in the cloud.

Acknowledgements

We thank our shepherd Amar Phanishayee and all the anonymous reviewers for all their valuable suggestions. Work on this article was funded via NWO VIDI MagnaData (#14826), SURFsara e-infra180061, as well as NSF Grant numbers CNS-1419199, CNS-1743363, OAC-1836650, CNS-1764102, CNS-1705021, OAC-1450488, and the Center for Research in Open Source Software.

Appendix – Code and Data Artifacts

Raw Cloud Data:
DOI:10.5281/zenodo.3576604
Bandwidth Emulator:
github.com/alexandru-uta/bandwidth_emulator
Cloud Benchmarking:
github.com/alexandru-uta/measure-tcp-latency

References

[1] *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017.

[2] *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 2018.

[3] A. Abedi and T. Brecht. Conducting repeatable experiments in highly variable cloud computing environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 287–292. ACM, 2017.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[5] A. C. Arpaci-Dusseau and G. Voelker, editors. *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 2018.

[6] AWS Enhanced Networking. `https://aws.amazon.com/ec2/features/#enhanced-networking`, 2019.

[7] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 242–253. ACM, 2011.

[8] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. There goes the neighborhood: performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 41. ACM, 2013.

[9] A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, and L. V. Kale. Identifying the culprits behind network congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 113–122. IEEE, 2015.

[10] S. M. Blackburn, A. Diwan, M. Hauswirth, P. F. Sweeney, J. N. Amaral, T. Brecht, L. Bulej, C. Click, L. Eeckhout, S. Fischmeister, et al. The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(4):15, 2016.

[11] J.-Y. L. Boudec. *Performance Evaluation of Computer and Communication Systems*. EFPL Press, 2011.

[12] B. Briscoe and M. Sridharan. Network performance isolation in data centres using congestion exposure (ConEx). IETF draft, 2012.

[13] Z. Cao, V. Tarasov, H. P. Raman, D. Hildebrand, and E. Zadok. On the performance variation in modern storage stacks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 329–344, 2017.

[14] G. Casale and M. Tribastone. Modelling exogenous variability in cloud deployments. *ACM SIGMETRICS Performance Evaluation Review*, 2013.

[15] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan. Scaling Spark on HPC systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 97–110. ACM, 2016.

[16] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[17] C. Curtsinger and E. D. Berger. Stabilizer: Statistically sound performance evaluation. *SIGARCH Comput. Archit. News*, 41(1):219–228, Mar. 2013.

[18] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 373–387, 2018.

[19] Databricks Project Tungsten. `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`.

[20] Databricks Instance Types. `https://databricks.com/product/aws-pricing/instance-types`, 2019.

[21] A. B. De Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney. Why you should care about quantile regression. In *ACM SIGPLAN Notices*, volume 48, pages 207–218. ACM, 2013.

[22] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[23] D. A. Dickey and W. A. Fuller. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American Statistical Association*, 74(366a):427–431, 1979.

[24] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 20. ACM, 2012.

[25] B. Ghit and D. Epema. Reducing job slowdown variability for data-intensive workloads. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2015.

[26] J. D. Gibbons and S. Chakraborti. *Nonparametric statistical inference*. Springer, 2011.

[27] Y. Gong, B. He, and D. Li. Finding constant from change: Revisiting network performance aware optimizations on iaas clouds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 982–993. IEEE Press, 2014.

[28] Google Andromeda Networking. https://cloud.google.com/blog/products/networking/google-cloud-networking-in-depth-how-andromeda-2-2-enables-high-throughput-vms, 2019.

[29] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, pages 1–14, 2015.

[30] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International Co-NEXT Conference*, page 15. ACM, 2010.

[31] T. Hoefler and R. Belli. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015.

[32] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.

[33] B. Hubert, T. Graf, G. Maxwell, R. van Mook, M. van Oosterhout, P. Schroeder, J. Spaans, and P. Larroy. Linux advanced routing & traffic control. In *Ottawa Linux Symposium*, page 213, 2002.

[34] A. Iosup, A. Uta, L. Versluis, G. Andreadis, E. Van Eyk, T. Hegeman, S. Talluri, V. Van Beek, and L. Toader. Massivizing computer systems: a vision to understand, design, and engineer computer ecosystems through and beyond modern distributed systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1224–1237. IEEE, 2018.

[35] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 104–113. IEEE, 2011.

[36] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990.

[37] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. The popper convention: Making reproducible systems evaluation practical. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1561–1570. IEEE, 2017.

[38] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. *SIGPLAN Not.*, 48, 2013.

[39] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 158–169. ACM, 2015.

[40] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010.

[41] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan. Choreo: Network-aware task placement for cloud applications. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 191–204. ACM, 2013.

[42] P. Leitner and J. Cito. Patterns in the chaos–a study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology (TOIT)*, 16(3):15, 2016.

[43] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 1–14. ACM, 2010.

[44] J. R. Lorch and M. Yu, editors. *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*. USENIX Association, 2019.

[45] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. Wild. Modeling I/O performance variability using conditional variational autoencoders. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 109–113. IEEE, 2018.

[46] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, pages 50–60, 1947.

[47] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 409–425, 2018.

[48] J. C. Mogul and L. Popa. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review*, 42(5):44–48, 2012.

[49] R. O. Nambiar and M. Poess. The making of TPC-DS. In *VLDB*, 2006.

[50] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *NSDI '15*, volume 15, pages 293–307, 2015.

[51] V. Persico, P. Marchetta, A. Botta, and A. Pescapé. Measuring network throughput in the cloud: the case of amazon ec2. *Computer Networks*, 93:408–422, 2015.

[52] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. *ACM SIGCOMM Computer Communication Review*, 37(4):337–348, 2007.

[53] J. Rexford, M. Balazinska, D. Culler, and J. Wing. Enabling computer and information science and engineering research and education in the cloud. 2018.

[54] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. O. Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *WIOV*, 2011.

[55] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, 2010.

[56] S. Shapiro and M. Wilk. An analysis of variance test for normality (complete samples). *Biometricka*, 52:591–611, Dec. 1965.

[57] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, pages 513–527, 2015.

[58] A. Trivedi, P. Stuedi, J. Pfefferle, A. Schuepbach, and B. Metzler. Albis: High-performance file format for big data systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 615–630, 2018.

[59] A. Uta, A. Custura, D. Duplyakin, I. Jimenez, J. Rellermeyer, C. Maltzahn, R. Ricci, and A. Iosup. Cloud Network Performance Variability Repository. `https://zenodo.org/record/3576604#.XfeXa0uxXOQ`, 2019.

[60] A. Uta and H. Obaseki. A performance study of big data workloads in cloud datacenters with network variability. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018.

[61] A. J. Viera, J. M. Garrett, et al. Understanding interobserver agreement: the kappa statistic. *Fam med*, 37(5):360–363, 2005.

[62] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis. Using burstable instances in the public cloud: Why, when and how? *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):11, 2017.

[63] G. Wang and T. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM*. IEEE, 2010.

[64] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[65] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12*. USENIX, 2012.

Learning Relaxed Belady for Content Distribution Network Caching

Zhenyu Song
Princeton University

Daniel S. Berger
Microsoft Research & CMU

Kai Li
Princeton University

Wyatt Lloyd
Princeton University

Abstract

This paper presents a new approach for caching in CDNs that uses machine learning to approximate the Belady MIN (oracle) algorithm. To accomplish this complex task, we propose a CDN cache design called Learning Relaxed Belady (LRB) to mimic a Relaxed Belady algorithm, using the concept of Belady boundary. We also propose a metric called good decision ratio to help us make better design decisions. In addition, the paper addresses several challenges to build an end-to-end machine learning caching prototype, including how to gather training data, limit memory overhead, and have lightweight training and prediction.

We have implemented an LRB simulator and a prototype within Apache Traffic Server. Our simulation results with 6 production CDN traces show that LRB reduces WAN traffic compared to a typical production CDN cache design by 4–25%, and consistently outperform other state-of-the-art methods. Our evaluation of the LRB prototype shows its overhead is modest and it can be deployed on today's CDN servers.

1 Introduction

Content Distribution Networks (CDNs) deliver content through a network of caching servers to users to improve latency, bandwidth, and availability. CDNs delivered 56% of Internet traffic in 2017, with a predicted rise to 72% by 2022 [29]. Improving CDN caching servers can significantly improve the content delivery of the Internet.

A CDN cache sits between users and Wide-Area Networks (WANs). Whenever a user requests content that is not currently cached, the CDN must fetch this content across Internet Service Providers (ISPs) and WANs. While CDNs are paid for the bytes delivered to users, they need to pay for the bandwidth required to fetch cache misses. These bandwidth costs constitute an increasingly large factor in the operational cost of large CDNs [6, 14, 17, 41, 46, 66]. CDNs are thus seeking to minimize these costs, which are typically measured as *byte miss ratios*, i.e., the fraction of bytes requested by users that are not served from cache.

The algorithm that decides which objects are cached in each CDN server plays a key role in achieving a low byte miss ratio. Yet, the state-of-the-art algorithms used in CDN caching servers are heuristic-based variants of the Least-Recently-Used (LRU) algorithm (Section 2). The drawback of heuristics-based algorithms is that they typically work well for some access patterns and poorly for others. And, even with five decades of extensive study since caching was first proposed [84]—including using machine learning to adapt heuristics to different workloads—the fundamental limitation of heuristics remains. We still observe a large gap between the byte miss ratios of the state-of-the-art online cache replacement algorithms and Belady's offline MIN algorithm [16] on a range of production traces.

To bridge this gap, this paper presents a novel machine-learning (ML) approach that is fundamentally different from previous approaches to cache replacement. Our approach does not build on heuristics, nor try to optimize heuristics-based algorithms. Our approach is to approximate Belady's MIN (oracle) algorithm, using machine learning to find objects to evict based on past access patterns. Belady's algorithm always evicts the object with the furthest next request. A naive ML algorithm that imitates this behavior would incur prohibitively high computational cost. To overcome this challenge, our key insight is that it is sufficient to approximate a *relaxed Belady algorithm* that evicts an object whose next request is beyond a threshold but not necessarily the farthest in the future.

To set a proper threshold, we introduce the *Belady boundary*, the minimum time-to-next-request of objects evicted by Belady's MIN algorithm. We show that the byte miss ratio of the relaxed Belady algorithm using the Belady boundary as its threshold is close to that of Belady's MIN algorithm. Approximating relaxed Belady gives our system a much larger set of choices to aim for, which has two major consequences for our design. It allows our system to run predictions on a small sampled candidate set—e.g., 64—which dramatically reduces computational cost. And, it allows our system to quickly adapt to changes in the workload by gathering training data that includes the critical examples of objects that relaxed Belady would have selected for replacement.

Even with the insight of relaxed Belady, the design space of potential ML architectures (features, model, loss function, etc.) is enormous. Exploring this space using end-to-end metrics like byte miss ratio is prohibitively time-consuming because they require full simulations that take several hours for a single configuration. To enable our exploration of the ML design space we introduce an eviction decision quality metric, the *good decision ratio*, that evaluates if the next request of an evicted object is beyond the Belady boundary. The good decision ratio allows us to run a simulation only once to gather

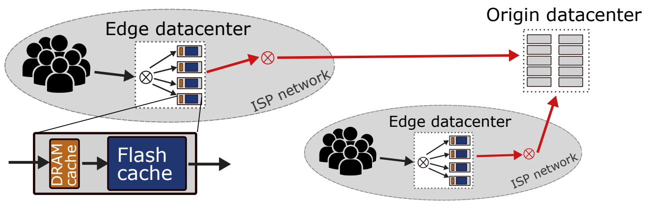


Figure 1: CDNs place servers in user proximity around the world (e.g., in a user's ISP). Incoming requests are sharded among several caching servers, which use combined DRAM and flash caches. Cache misses traverse expensive wide-area networks to retrieve data from the origin datacenter.

training data, prediction data, and learn the Belady boundary. Then we can use it to quickly evaluate the quality of decisions made by an ML architecture without running a full simulation. We use the good decision ratio to explore the design space for components of our ML architecture including its features, model, prediction target, and loss function among others.

These concepts enable us to design the Learning Relaxed Belady (LRB) cache, the first CDN cache that approximates Belady's algorithm. Using this approach in a practical system, however, requires us to address several systems challenges including controlling the computational overhead for ML training and prediction, limiting the memory overhead for training and prediction, how to gather online training data, and how to select candidates for evictions.

We have implemented an LRB simulator and an LRB prototype within Apache Traffic Server [1, 2, 10, 43].[1] Our evaluation results using 6 production CDN traces show that LRB consistently performs better than state-of-the-art methods and reduces WAN traffic by 4–25% compared to a typically deployed CDN cache—i.e., using LRU replacement that is protected with a Bloom filter [22]. Our evaluation of the prototype shows that LRB runs at a similar speed to a heuristic-based design and its computational overhead and memory overhead are small.

2 Background and Motivation

In this section, we quantify the opportunities and challenges that arise in CDN caching. We also discuss the constraints on the design space of LRB.

2.1 The Challenge of Reducing WAN Traffic

A CDN user request is directed to a nearby CDN server (e.g., using DNS or anycast [33]), which caches popular web and media objects. Each cache miss has to be fetched across the wide-area network (WAN) (Figure 1). As WAN bandwidth needs to be leased (e.g., from tier 1 ISPs), CDNs seek to

[1]The source code of our simulator and prototype alongside with documentation and the Wikipedia trace used in our evaluation is available at `https://github.com/sunnyszy/lrb` .

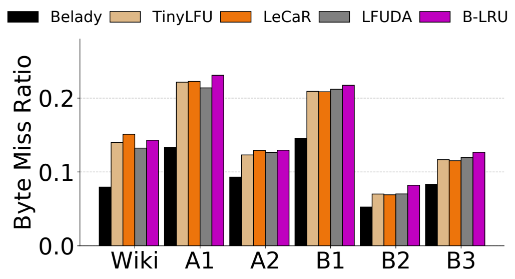


Figure 2: Simulated byte miss ratios for Belady and the top-performaning policies from Section 6 for six CDN production traces and a 1 TB flash cache size. There is a large gap of 25–40% between Belady and all other policies.

reduce WAN traffic, which corresponds to minimizing their cache's byte miss ratio.

CDN caching working sets are massive compared to available cache space. To reduce bandwidth, a CDN deploys DRAM and flash caches in each edge server.[2] A commercial CDN serves content on behalf of thousands of content providers, which are often large web services themselves. This leads to a massive working set, which is typically sharded across the (tens of) servers in an edge cluster [25, 77]. Even after sharding traffic, an individual cache serves traffic with a distinct number of bytes much larger than its capacity. Consequently, there exists significant competition for cache space.

There are significant opportunities to improve byte miss ratios. CDN traffic has a variety of unique properties compared to other caching workloads. For example, a striking property observed in prior work [63] is that around 75% of all objects do not receive a second request within two days (so-called "one-hit-wonders"). If these objects are allowed to enter the cache, the vast majority of cache space is wasted. Therefore, major CDNs deploy B-LRU, an LRU-eviction policy using a Bloom filter [63, 67, 74] to prevent one-hit-wonder objects from being admitted to the cache.

We quantify the opportunity to improve byte miss ratios over B-LRU. To this end, we use 6 production traces from 3 different CDNs, which are sharded at the granularity of a single SSD (around 1–2 TB). Figure 2 compares the byte miss ratios of B-LRU and the other top-performing policies from Section 6 to Belady's MIN [16], and we find that there remains a gap of 25–40%.

Exploiting miss ratio opportunities is challenging in practice. The key challenge is that workloads change rapidly over time and differ significantly between servers and geographic locations. These changes occur due to load balancing different types of content, e.g., web traffic is more popular in the morning and video more popular at night. Due to these rapid changes and breadth of different access patterns, prior

[2]CDNs typically prefer flash storage over HDDs as cache reads lead to random reads with very high IOPS requirements. While some CDNs have hybrid deployments, many CDNs rely entirely on flash [46, 67, 74, 78].

	PT-1	PD-2	ET-1	ET-2	CET-1	CET-2
Mean CPU	5%	3%	6%	16%	7%	18%
Peak CPU	19%	12%	10%	24%	13%	30%

Table 1: Mean and peak CPU load in Wikipedia's production deployment CDN across 6 datacenters in three different timezones, for March 2019. Peak CPU load is below 30%.

caching policies (Section 7) only achieve marginal improvements over B-LRU (Section 6). In fact, many recent caching policies lead only to gains on some traces with certain cache size configurations.

In conclusion, there is significant demand to realize caching policies that can automatically adapt to a geographic location's workload and to changes in request patterns over time. Machine learning is well suited to achieving this goal. But, leveraging machine learning for CDN caching requires us to overcome many challenges we describe in Section 4.

2.2 Opportunity and Requirements

Deployment opportunity. We find that today's CDN caches typically have spare processing capacity. Table 1 shows the CPU load in six production deployments of Wikipedia's CDN for March 2019. We see that, on average, there is 90% spare processing capacity. Even under peak load, there is still 70% CPU capacity available, as disk and network bandwidth are frequent bottlenecks. Thus, the opportunity in current deployment is that we can use this excess processing capacity as part of more sophisticated caching algorithms.

Deployment requirements. There are three key constraints when deploying on existing CDN caching servers.

- **Moderate memory overhead** of a few GBs but not 10s of GBs because large quantities are not available [19].
- **Not require TPUs or GPUs** because these are not currently deployed in CDN servers [41, 63].
- **Handle tens of thousands of requests per second** because this is the request rate seen at CDN caches [19].

3 Approximating Belady's MIN Algorithm

In order to approximate Belady's MIN algorithm in the design of an ML-based cache, this section introduces the relaxed Belady algorithm, Belady boundary, and good decision ratio.

3.1 Relaxed Belady Algorithm

It is difficult for an ML predictor to directly approximate Belady's MIN algorithm for two reasons. First, the cost of running predictions for all objects in the cache can be prohibitively high. Second, in order to find the object with the farthest next request, an ML predictor needs to predict the time to next request of all objects in the cache accurately.

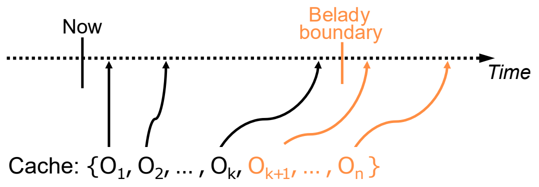


Figure 3: The relaxed Belady algorithm partitions objects into two sets based on their next requests. The next requests of $O_{k+1}, \ldots, O_n$ are beyond the threshold (Belady boundary).

To overcome these two challenges, we define the *relaxed Belady algorithm*, a variant of Belady's MIN, that randomly evicts an object whose next request is beyond a specific threshold, as shown in Figure 3. The algorithm partitions the objects of a cache into two sets each time it needs to make an eviction decision: objects whose next requests are within the threshold (set 1) and objects whose next requests are beyond the threshold (set 2). If set 2 is not empty, the algorithm randomly evicts an object from this set. If set 2 is empty, the algorithm reverts to the classical Belady among object in set 1.

The relaxed Belady algorithm has two important properties. First, if the threshold is far, its byte miss ratio will be close to that of Belady's MIN. Second, since the algorithm can evict any of the objects in set 2, not necessarily the object with farthest distance, there will be many eviction choices in the cache each time it needs to evict an object.

These properties greatly reduce the requirements for an ML predictor. Instead of predicting next requests for all objects in the cache, the ML predictor can run predictions on a small sample of candidates in the cache for an eviction as long as the sample includes objects from set 2. This allows a dramatic reduction of computational cost. In addition, it reduces the required precision of the ML predictor. It only needs to find an object whose next request is beyond the threshold instead of the farthest. This greatly reduces the memory overhead when gathering training data: instead of having to track objects indefinitely, it is sufficient to track objects until they are re-requested or cross the threshold.

Shorter thresholds thus make the task of the ML predictor more tractable. While longer thresholds move the byte miss ratio of relaxed Belady closer to Belady's MIN. It is thus important to find the proper threshold.

3.2 Belady Boundary

To systematically choose a threshold for the relaxed Belady algorithm, we introduce the *Belady boundary* as the minimum of time to next request for all evicted objects by Belady's MIN algorithm. It is intuitively meaningful—Belady's MIN kept all objects with a next request less than this boundary in the cache—but requires future information to compute. In practice, we assume that the Belady boundary is approximately stationary. This allows us to approximate the Belady boundary by computing the minimum of the next requests of

Cache size (GB)	64	128	256	512	1024
Boundary (x cache size)	3	2	2	2	2
Objects beyond boundary	19%	19%	16%	12%	10%
Increase in byte miss ratio over Belady	10%	13%	12%	11%	9%

Table 2: Comparison of the relaxed Belady algorithm to Belady MIN for the Wikipedia trace with different cache sizes.

all evicted objects by the Belady MIN algorithm during the machine learning warmup period.

Table 2 shows how the Belady boundary affects the byte miss ratio of relaxed Belady on the Wikipedia trace. We find that relaxed Belady applies random eviction to a set of objects, e.g., 19% for a 64 GB cache. While this Belady boundary enables efficient implementations, it comes at the cost of 9-13% more misses. Compared to the 25%–40% gap between state-of-the-art online algorithms and Belady's MIN (Figure 2), this increase seems acceptable.

Next, we show how to use the Belady boundary to establish a decision quality metric, which is used to make design decisions throughout Section 4.

3.3 Good Decision Ratio

To design an algorithm that makes better decisions we need to determine the quality of individual decisions. End-to-end metrics like byte miss ratio, however, reflect the aggregated quality of many decisions. When an algorithm has a byte miss ratio higher than Belady we know it made some bad decisions, but we cannot determine which of its decisions were bad. The individual misses that comprise byte miss ratio are similarly unhelpful because they are also the result of many earlier decisions.

Our eviction decision metric is defined with respect to the relaxed Belady algorithm with the Belady boundary as its threshold. Evicting an object is a *good decision* if the next request of the object is beyond the Belady boundary. It is a *bad decision* if its next request is within the Belady boundary.

We find that an algorithm's *good decision ratio*—i.e., # good decisions / # total eviction decisions—correlates strongly with the byte miss ratio it ultimately achieves (Section 6.4). This metric plays a key part in the design of LRB.

4 Design of Learning Relaxed Belady Cache

This section presents the design details of LRB, which uses ML to imitate the relaxed Belady algorithm. Accomplishing this requires simultaneously addressing two previously unsolved problems:

- How to design an ML approach that decreases byte miss ratio relative to state-of-the-art heuristics?

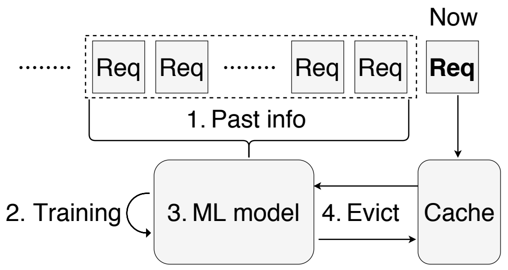


Figure 4: General architecture that uses ML for caching with 4 key design issues: 1) what past information to use for ML, 2) how to generate online training datasets, 3) what ML model to use, and 4) how to select eviction candidates.

- How to build a practical system with that approach?

Each problem introduces several challenges in isolation, simultaneously addressing them additionally requires us to balance their often-competing goals.

Figure 4 presents a general architecture that uses ML to make eviction decisions. We identify four key design issues:

(1) Past information. The first design issue is determining how much and what past information to use. More data improve training quality, but we need to limit the information in order to build a practical system, as higher memory overhead leads to less memory that can be used to cache objects.

(2) Training data. The second design issue is how to use past information for training. As workloads vary over time the model must be periodically retrained on new data. Thus, a practical system must be able to dynamically generate training datasets with proper labels.

(3) ML architecture. The third design issue is selecting a machine learning architecture that makes good decisions. This includes selecting features, the prediction target, and the loss function as well as how to translate predictions into eviction decisions. In addition, these decisions need to be compatible with available hardware resources to build a practical system.

(4) Eviction candidate selection. The final design issue is how to select eviction candidates. Although an approximation of the relaxed Belady algorithm can evict any object whose next request is beyond the Belady boundary and there are many eviction choices, we still need a method to select a small set of candidates that includes such objects with a high probability.

The rest of this section describes the design decisions of LRB for each of these design issues. Our general approach is to guide our design using the good decision ratio metric defined in Section 3.

4.1 Past information

LRB keeps information about an object only when its most recent request is within the *sliding memory window*. The information within the sliding memory window is used for training and prediction.

Setting the size of the sliding memory window is important to the performance of LRB. If the sliding memory window is too short, LRB will not have enough training data to produce a good model and may not have the information it needs during prediction to make an accurate prediction. If the sliding memory window is too long, it may take too much space away from cached data.

LRB uses each trace's validation prefix to set the sliding memory window hyperparameter. For small cache sizes, the validation prefix is long enough to derive the "optimal" sliding memory window, which achieves the highest good decision ratio. For large cache sizes, we use a least-squares regression line fit to the relationship between small cache sizes and their optimal sliding memory window parameters.

We remark that, at the beginning of a trace (or during initial deployment), LRB requires training data to build an initial model. Thus, our implementation uses LRU as a fallback until sufficient training data is available.

4.2 Training Data

Acquiring training data for our problem is challenging because the features and corresponding label exist at widely disparate times. Consider an object that is requested once and then not requested again until 5 hours later. The features of the object exist and vary at all times in the 5-hour range. The label—i.e., what we want to predict—does not exist until the end of that range, when we know the "future" by waiting until it is the present. To address the time disparity, LRB decouples generation of unlabeled training data from labeling that data.

Unlabeled training data generation. To generate unlabeled training data LRB periodically takes a random sample of objects in the sliding memory window and then records the current features of those objects. We choose to randomly sample over objects instead of over requests to avoid biasing the training data towards popular objects with many requests. Such popular objects should be represented in the training data so the model learns not to evict them. Training data for less popular objects, however, is more important because they include the objects beyond the Belady boundary—the good eviction decisions we want our model to learn. We choose to randomly sample over all objects in the sliding memory window instead of only those currently in the cache as cached objects are similarly biased.

Labeling training data. LRB uses two methods for assigning labels to training data. The first is to wait until an object is requested again. When this happens, we know the "future" and use that to determine the label. Some objects, however, will not be requested until far—e.g., 5 days—in the future.

Waiting until such objects are requested again would introduce many problems. First, it would require excessive memory overhead as we maintain features for many objects that were requested a potentially very long time ago. Second, it would make some of the training data excessively old, e.g., 5 days old. Finally, it would never include training data for objects that are never requested again—which are a significant portion of the good decisions we want our method to learn.

LRB's second method for labeling training data avoids these problems by leveraging an insight related to the Belady boundary: all objects that are not requested again for at least a boundary amount of time are equivalent good eviction decisions. Thus, once the time since last request for an object exceeds the Belady boundary we can safely label it. LRB uses this second labeling method when an object falls out of the sliding memory window.

4.3 ML Architecture

This subsection describes the components of LRB's ML architecture. For each component, we describe our choice, describe its rationale, and then use the good decision ratio to explore the design space. This exploration is complicated by the fact that each component influences the others—feature set A may be better for model X while feature set B may be better for model Y. Even with our good decision ratio doing a full exploration of all possible combinations is still prohibitive. Instead, we fix each of the other components on what we ultimately select in LRB's design and then vary the specific component.

4.3.1 Features: Deltas, EDCs, and Static

LRB uses three different types of features: deltas, exponentially decayed counters, and static features. The rationale for these features is to provide a superset of the information used by previous heuristics. Because our model learns the weight for different features adding more features should not harm its accuracy. Thus, the primary tradeoff for features is weighing how much they help the model against the overhead they incur. To minimize the overhead of the features we strive to make storing and updating them efficient (Section 5).

Deltas. Deltas indicate the amount of time between consecutive requests to an object. Delta$_1$ indicates the amount of time since an object was last requested. Delta$_2$ indicates the time in between an object's previous two requests and so on, i.e., delta$_n$ is the amount of time between an object's n^{th} and $(n-1)^{th}$ previous requests. If an object has been requested only n times, then above delta$_n$ are ∞. We include deltas as features because they subsume the information used by many successful heuristics. For instance, LRU uses delta$_1$, LRU-K uses the sum of delta$_1$ to delta$_k$, and S4LRU uses a combination of delta$_1$ to delta$_4$.

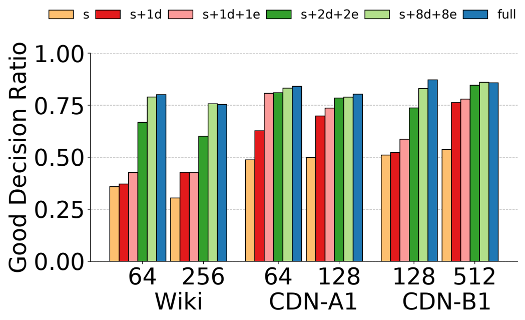


Figure 5: Good decision ratios for accumulating features for LRB on three traces at two cache sizes. LRB uses static features, 32 deltas, and 10 EDCs (full).

Exponentially decayed counters (EDCs). EDCs track an approximate per-object request count over longer time periods, where keeping exact counts would require excessive memory overhead. Each EDC$_i$ is initialized to 0. Whenever there is a request to the object, we first update Delta$_1$ and then $C_i = 1 + C_i \times 2^{-\text{Delta}_1/2^{9+i}}$. Unlike Delta$_1$, C_i will not be updated until another request arrives for the same object. The EDCs with larger values of i cover larger periods, e.g., an object that was highly popular 1 million requests ago but is now not popular would have a low EDC$_1$ but still have a high EDC$_{10}$.

The exact EDC equation is motivated by prior observation in block storage caching [57] and video popularity prediction [79], where EDCs accurately approximate the decay rate of object popularities. Tracking long-term popularity is used by many other algorithms such as LFU variants [11,75], Hyperbolic [21], and LRFU [57] (which uses a single EDC). LRB uses multiple EDCs with different decay constants to capture the request rate to the object over multiple time horizons.

Static features. Static features include additional unchanging information about an object. They include the size of the object and its type—e.g., video on demand segment, live video segment, image. We include static features because they are available, require little memory overhead, and intuitively correlate with different access patterns. For instance, live video segments might be useful in a cache for only a short period, whereas video on demand segments might be useful in the cache for much longer.

Feature effectiveness, number of deltas, number of EDCs. To determine if this set of features is effective for our ML architecture we evaluate them using the good decision ratio in the validation prefix of our traces at many cache sizes. We also use the good decision ratio to determine how many deltas we store and how many EDCs we store. Figure 5 shows the results for an accumulation of static features, Delta$_1$, EDC$_1$, Delta$_2$ and EDC$_2$, Deltas 3–8 and EDCs 3—8, and then Deltas 9–32 and EDCs 9–10 (full). As expected, the addition of more features improves the good decision ratio but has diminishing returns. Results on the other three traces and at other cache sizes are not shown but are similar.

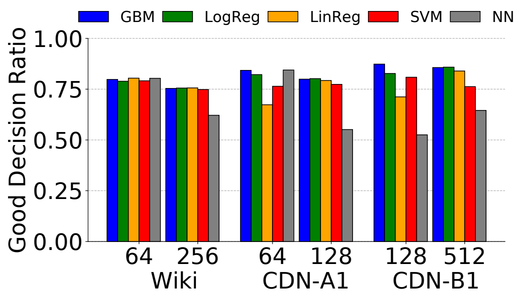


Figure 6: Good decision ratios for models. LRB uses GBM as it robustly achieves high good decision ratios on all traces/cache sizes.

The number of deltas and EDCs is each a tradeoff between giving the ML architecture more information to use for decisions and storage overhead. We settle on 32 deltas and 10 EDCs because both appear to be past the point of diminishing returns and thus we are not disadvantaging our ML architecture by stopping there. We go as high as these numbers because of the efficiency of our feature storage and our ability to control memory overhead with the sliding memory window.

4.3.2 Model: Gradient Boosting Machines

LRB uses Gradient Boosting Machines [39] (GBM) for its model, which outperform all other models we explored and are highly efficient on CPUs. We were inspired to explore GBM based on their success in many other domains with tabular data [23,30,42,61,62,72,80]. We also explored linear regression, logistic regression, support-vector machines, and a shallow neural network with 2 layers and 16 hidden nodes. Figure 6 shows the good decision ratios of the different models on three traces at two cache sizes. Results on the other traces and other cache sizes are similar and not shown. GBM robustly achieve a high good decision ratio. Additionally, GBM do not require feature normalization and can handle missing values efficiently, which are common as objects have a varying number of Deltas. In addition, GBM are highly efficient to train and use for prediction. On typical CDN server hardware, we can train our model in 300 ms. And, we can run prediction on 64 eviction candidates in 30 μs.

4.3.3 Prediction Target: log (time-to-next-request)

LRB uses regression to predict the time-to-next-request for objects that are requested within the sliding memory window. Regression enables LRB to rank these objects (as relaxed Belady) and also serves as a confidence measure for distance to the Belady boundary. Specifically, LRB predicts the log(time-to-next-request) for an object as we primarily care about which side of the boundary the next request is on. If the Belady boundary is 1M (measured by the number of requests), then the difference between predicting 100K and 2M

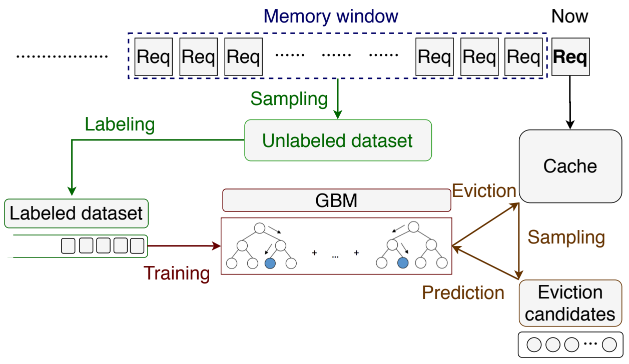


Figure 7: Detailed architecture overview of LRB.

is larger than the difference between 2M and 3M. For objects that are not requested within the sliding memory window, LRB assigns a label as 2× the window size.

We also explored predicting the time from the last request to the next request, binary classification relative to a boundary, and unmodified time-to-next-request. We chose log(time-to-next-request) because it achieved a higher good decision ratio than all of these alternatives.

4.3.4 Loss Function: L2

LRB uses L2 loss (mean square error). We also tried all eight loss functions in the LightGBM library [54]. We chose L2 loss because it achieved the highest good decision ratio.

4.3.5 Training dataset Size: 128K

LRB trains a new model once it accumulates a dataset of 128K labeled training samples. We also explored training dataset sizes that were much smaller than 128K and up to 512K. We found that good decision ratio increases with dataset size but has diminishing returns. We choose 128K because larger dataset sizes further increase training time and overhead without a noticeable improvement in good decision ratio.

4.4 Eviction Candidate Selection.

LRB randomly samples cached objects to gain eviction candidates and runs a batch prediction for all candidates. LRB evicts the candidate whose predicted next request distance is the longest.

We determine the choice for our random sample size—64 samples—using the good decision ratio. We find that 64 samples are past the point of diminishing returns, and thus choosing it does not disadvantage our ML architecture. It is also still low enough for low overhead—prediction on 64 samples takes LRB only 30 μs.

4.5 Putting All Together

Putting our design decisions together with the general architecture (Figure 4) from the beginning of the section, we have the complete design of LRB as shown in Figure 7.

LRB learns from the requested objects in a sliding memory window whose length approximates the Belady boundary. The features (deltas, EDCs, static) of these objects are stored in a compact data structure that is updated as new requests arrive, moving the sliding memory window forward.

A sampling process continuously samples data from this data structure, generating an unlabeled dataset. A separate labeling process continuously labels that data. When the labeled dataset is full (128K examples), LRB starts training a GBM model, and empties the labeled dataset. After that, whenever the labeled dataset is full again, LRB repeats the process and replaces the old model with the new one. If a current request is a cache miss and needs to evict an object, LRB randomly draws $k = 64$ eviction candidates and runs GBM predictions on them. LRB evict the candidate with the farthest predicted next access time.

5 Implementation

We have implemented a LRB prototype within Apache Traffic Server (ATS). We have also implemented a LRB simulator in order to compare with a wide range of other algorithms. The two implementations share the same code and data structures as a C++ library with about 1400 lines of code.

5.1 Prototype

ATS is a multi-threaded, event-based CDN caching server written in C++. A typical ATS configuration consists of a memory cache and a disk/SSD cache. ATS uses a space-efficient in-memory lookup data structure as an index to the SSD cache, which is accessed using asynchronous I/Os to achieve high performance.

To implement LRB, we have replaced the lookup data structures for ATS's disk cache with the architecture described in Section 4.5. We have also changed ATS to make eviction decisions asynchronously by scheduling cache admissions in a lock-free queue. A known challenge when implementing SSD-based caches is write amplification due to random writes [36, 58]. In order to implement LRB, a production implementation would rely on a flash abstraction layer such as RIPQ [78] or Pannier [58]. Unfortunately, we have no access to such flash abstraction layers (e.g., RIPQ is proprietary to Facebook). Our implementation thus emulates the workings of a flash abstraction layer, reading at random offsets and writing sequentially to the SSD.

As the memory cache is typically small, which has a negligible impact on the byte miss ratio [19], we leave this part of

Object class: # past requests	1	2	3	4	5	6-12	13-31	$\geq$32
Obj fraction (%)	36	11	5	3	2	1	<1	<1
Overhead (bytes)	25	94	98	102	106	$\leq$134	$\leq$210	214

Table 3: LRB's memory overhead depends on an object's number of past requests, which we call the object's class.

ATS unchanged. In total, the changes to ATS, excluding the LRB library, amount to fewer than 100 lines of code.

5.2 Simulator

We have implemented an LRB simulator based on the AdaptSize simulator [19]. Besides LRB, our simulator implements 14 state-of-the-art caching algorithms.[3] These include classic and learning-based algorithms. The classic algorithms include LRUK [68], LFUDA [11, 75], S4LRU [46], LRU, FIFO, Hyperbolic [21], GDSF [11] and GDWheel [60]. The learning-based algorithms include an adaptive version of TinyLFU [34] (which subsumes static TinyLFU [35]), LeCaR [81], UCB [31] (reinforcement learning), LFO [17] (supervised learning), LHD [15] and AdaptSize [19]. These 14 algorithms and the simulation environment take about 11K lines of C++. For TinyLFU, LFO, LHD, and AdaptSize, we verified parameter configurations match the authors'.

5.3 Optimizations

We have implemented two main optimizations to reduce overhead and improve performance.

Computationally efficient feature updating. To minimize the overhead of maintaining features we seek to update them rarely and for necessary updates to be efficient. LRB accomplishes this by favoring *time-invariant* features when possible, i.e., features that need only be calculated once. Static features by definition fall in this category. We choose to make deltas relative to the time in between consecutive requests instead of the time between n requests ago and the most recent request because this makes all deltas other than delta_1 time-invariant. A new request to an object shifts delta_n to delta_{n+1}. The overhead to compute delta_1 is low and we make EDC updates efficient using a lookup table with precomputed decay rates.[4]

Overall, these optimizations result in a constant and small feature update overhead per request. LRB updates an object's feature only at the times when it is requested and when it is sampled as an eviction candidate.

We compress our features to minimize their memory overhead. Our compression is based on the predominance of "one-hit wonders" (Section 2). Consequently, we treat objects for which we registered only a single request separately. Since such an object was requested only once in the sliding memory window, there is no recency information and EDC values are all 0. Instead of keeping this redundant information, we store only the key, object size, object type, the last request time, and a pointer to a struct. This reduces the memory overhead for single-request objects to 25 B on the Wikipedia trace. When objects receive more requests, we populate the struct with only as many deltas as they have past requests. This further reduces the memory overhead for all objects with fewer requests than the maximum number of deltas, 32. The struct also contains the EDCs, which are compressed to a single float [57]. Table 3 shows the distribution and overhead of objects with different number of requests.

[3]For Adaptive-TinyLFU, we integrate the original author's implementation (`https://github.com/ben-manes/caffeine`) into our simulator.

[4]The size of the table is $W/2^{10}\times 4\,\text{B}$ where W is the sliding memory window size—e.g., if W is 2^{28} (256 million), the table is $2^{18}\times 4\,\text{B} = 1\,\text{MB}$.

6 Evaluation

This section evaluates our LRB prototype. We additionally use simulations to compare LRB to a wide range of state-of-the-art caching algorithms. We aim at answering the following questions:

- What is the WAN traffic reduction using our LRB prototype compared to the ATS production system (Sec 6.2)?
- What is the overhead of our LRB prototype compared to CDN production systems (Sec 6.3)?
- How does the byte miss ratio of LRB compare to state-of-the-art research systems on a wide range of CDN traces and cache sizes (Sec 6.4)?
- What is the gap between Belady and LRB (Sec 6.5)?
- Can LRB be improved using a longer validation prefix to select the sliding memory window (Sec 6.6)?

6.1 Experimental Methodology

This subsection describes the traces, competing cache algorithms, warm-up, and the testbed in our experiments.

Traces Our evaluation uses CDN traces from three CDNs, two of which chose to remain anonymous.

- **Wikipedia**: A trace collected on a west-coast node, serving photos and other media content for Wikipedia pages.
- **CDN-A**: Two traces (A1 and A2) collected from two nodes on different continents serving a mixture of web traffic for many different content providers.
- **CDN-B**: Three traces (B1–B3) collected from nodes in the same metropolitan area, each serving a different mixture of web and video traffic for many different content providers.

Table 4 summarizes key properties of the six traces.

State-of-the-art algorithms. In the prototype experiments, we compare our LRB implementation (Section 5) to unmodified Apache Traffic Server (ATS, version 8.0.3), which approximates a FIFO eviction policy. Our simulations compare

		Wikipedia	CDN-A1	CDN-A2	CDN-B1	CDN-B2	CDN-B3
Duration (Days)		14	8	5	9	9	9
Total Requests (Millions)		2,800	453	410	1,832	2,345	1,986
Unique Obj Requested (Millions)		37	89	118	130	132	116
Total Bytes Requested (TB)		90	156	151	638	525	575
Unique Bytes Requested (TB)		6	65	17	117	66	104
Warmup Requests (Millions)		2,400	200	200	1,000	1,000	1,000
Request Obj Size	Mean (KB)	33	644	155	460	244	351
	Max (MB)	1,218	1,483	1,648	1,024	1,024	1,024

Table 4: Summary of the six production traces that are used throughout our evaluation spanning three different CDNs.

LRB with 14 state-of-the-art caching algorithms (Section 5.2). In addition, our simulations include Belady's MIN [16] and relaxed Belady (Section 3) as benchmarks.

Testbed. Our prototype testbed consists of three Google cloud VMs acting as a client, CDN caching server, and backend/origin server, respectively. The VMs are n1-standard-64 VMs with 64 VCPUs, 240 GB of DRAM. To maximize SSD throughput, we use eight local 375 GB NVMe flash drives and combine them into one logical drive using software raid.

Clients are emulated using our C++ implementation ($\approx$ 200 LOC), which uses 1024 client threads. The backend/origin server uses our own concurrent C++ implementation ($\approx$ 150 LOC). This emulation method can saturate the network bandwidth between client and caching server. The clients replay requests in a closed loop to stress the system being tested and accelerate evaluation.

In Section 6.3, clients send requests in an open loop, using the original trace timestamps to closely emulate production workloads for latency measurements. Both unmodified ATS and LRB use a 1 TB flash cache size. Our simulations are based on the request order following the request timestamps and a range of different cache sizes.

To emulate network RTTs [19], we introduce around 10 ms and 100 ms of delays to the link between client and proxy and the link between origin and proxy respectively.

Metadata overhead. Different algorithms may have different metadata overheads. For fairness, all algorithms except Adaptive-TinyLFU [5] in the experiments have deducted their metadata overheads from corresponding cache sizes in all experiments. For example, if an experiment is for a 64 GB cache, an algorithm with 2 GB of metadata overhead will use 62 GB to store its data.

Validation and warmup trace sections. The first 20% of every trace is used as a "validation" section where LRB tunes its hyperparameters (Section 4). Each experiment uses a warmup during which no metrics are recorded. The warmup section is always longer than the validation section and is defined by the time by which every algorithm has achieved a stable byte miss ratio. Table 4 lists each trace's warmup section.

[5]Our experiments used an existing implementation of Adaptive-TinyLFU simulator which does not provide overhead information.

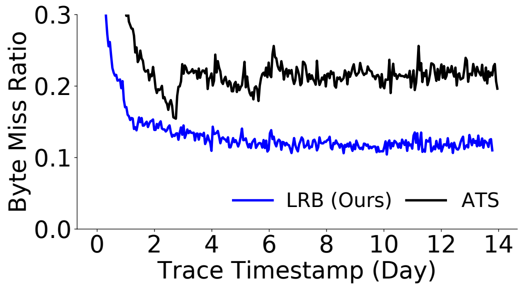


Figure 8: The byte miss ratios of LRB and unmodified ATS for the Wikipedia trace using a 1 TB cache size.

6.2 WAN Traffic Reduction of LRB Prototype

Figure 8 compares the byte miss ratios of LRB and unmodified ATS for a 1 TB cache using the Wikipedia trace. LRB achieves a better overall byte miss ratio than ATS. We observe that LRB reaches a lower miss ratio than ATS within the first day, and LRB continues to improve as it obtains more training data. Throughout the experiment, LRB's miss ratio is more stable than ATS's. We wait for both to stabilize and measure the WAN bandwidth consumption on days 12-14. LRB reduces the average bandwidth by 44% over ATS. LRB also reduces the 95^{th} percentile bandwidth (the basis of some CDN traffic contracts [6]) by 43% over ATS.

6.3 Implementation Overhead

Table 5 compares the overhead of LRB against unmodified ATS. We measure throughput, CPU, and memory utilization at peak throughput ("max" experiment). Note that Table 5 quantifies overheads for the Wikipedia trace, which is pes-

Metric	Experiment	ATS	LRB
Throughput	max	11.66 Gbps	11.72 Gbps
Peak CPU	max	9%	16%
Peak Mem	max	39 GB	36 GB
P90 Latency	normal	110 ms	72 ms
P99 Latency	normal	295 ms	295 ms
Obj Misses	normal	5.7%	2.6%

Table 5: Resource usage for ATS and LRB in throughput-bound (max) and production-speed (normal) experiments.

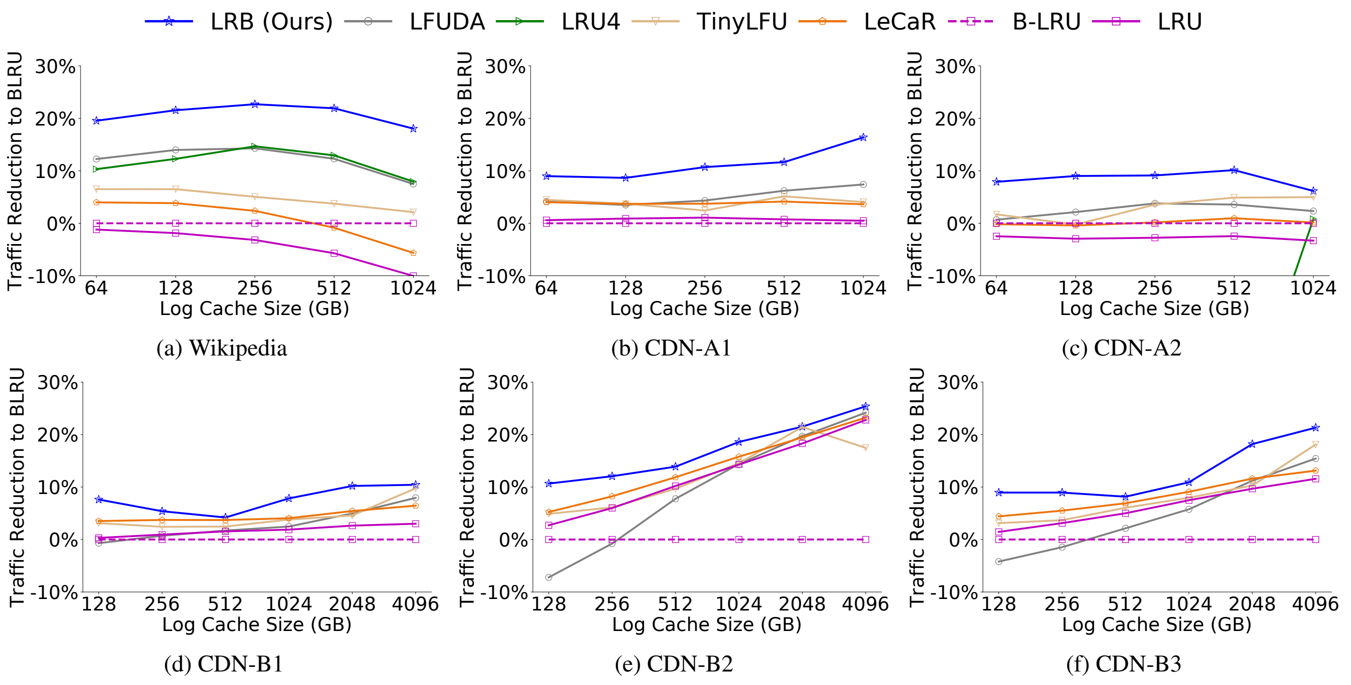


Figure 9: WAN traffic reduction over B-LRU at varying caches for Belady, LRB, and the six best state-of-the-art algorithms. LRB typically provides 4–25% WAN traffic savings over B-LRU.

simistic as all object sizes in all other traces are at least 5× larger, which results in significantly less overhead.

LRB has no measurable throughput overhead but its peak CPU utilization increases to 16% from 9% for ATS and 12% for B-LRU. We remark that even the busiest production cluster at Wikipedia ("CET-2" in Table 1 in Section 2) has sufficient CPU headroom.

We measure the number of object misses (which weights requests equally) when replaying the Wikipedia trace using its original timestamps ("normal" experiment in Table 5). LRB has less than half as many object misses as ATS. This miss reduction allows LRB to improve the 90^{th} percentile latency (P90) by 35% compared to ATS. At the 99^{th} percentile (P99), LRB achieves a similar latency to ATS because the origin server latency dominates.

Cache size	Wiki	A1	A2	B1	B2	B3
64 GB	3.0%	1.0%	1.7%	-	-	-
128 GB	2.1%	0.6%	1.4%	0.9%	0.8%	1.1%
256 GB	1.4%	0.5%	1.2%	0.8%	0.5%	0.6%
512 GB	1.0%	0.4%	1.0%	0.6%	0.4%	0.5%
1 TB	0.6%	0.3%	0.7%	0.5%	0.4%	0.4%
2 TB	-	-	-	0.4%	0.3%	0.3%
4 TB	-	-	-	0.3%	0.3%	0.3%

Table 6: Fraction of space allocated to metadata for LRB.

As LRB's memory overhead depends on the cache size and average object size in a trace, Table 6 measures the peak memory overhead for all traces and all evaluated cache sizes. Across all configurations, LRB always uses less than 3% of the cache size to store LRB metadata. While this overhead reduces the effective cache size, the small loss in byte miss ratio is more than offset by LRB's significant miss ratio improvements.

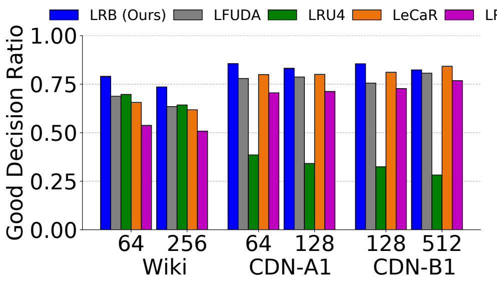


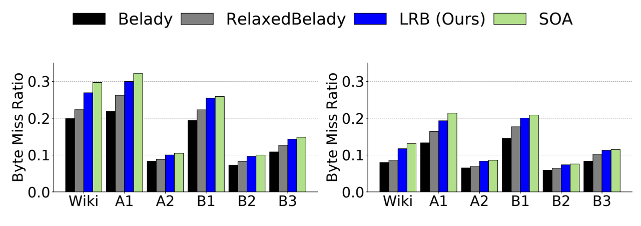
Figure 10: Good decision ratios for different algorithms. Good decision ratio correlates strongly with byte miss ratio.

We believe these experimental observations show that LRB is a practical design for today's CDNs and that it can be deployed on existing CDN hardware.

6.4 LRB vs. State-of-the-art Algorithms

We compare the byte miss ratio of LRB to 14 state-of-the-art caching algorithms using simulations with a wide range of cache sizes using the six different traces.

Of the 14 algorithms, nine algorithms (FIFO, Hyperbolic, GDSF, GDWheel, UCB, LFO, AdaptSize, S4LRU, and LHD)



(a) 256 GB (b) 1 TB

Figure 11: Comparison of byte miss ratios for Belady, relaxed Belady, LRB, and the best state-of-the-art (SOA) policy.

achieve a low byte miss ratio on at least one CDN traces. To improve readability, we show only the five best-performing algorithms in the following figures (TinyLFU, LeCaR, LRUK, LFUDA, and LRU). Figure 9 shows the wide-area network traffic reductions of each of these algorithms relative to B-LRU, with different cache sizes using the six traces.

LRB robustly outperforms the best state-of-the-art algorithms. It achieves the lowest byte miss ratio for all 33 CDN trace and cache size combinations. Overall, LRB reduces WAN traffic by 4–25% on average.

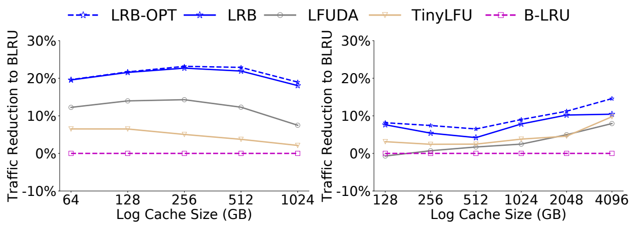
Note that LRB's WAN traffic reduction does not generally decrease with larger cache sizes. For example, on CDN-B2 the traffic reduction steadily increases between 128 GB and 4 TB. Our interpretation is that these traces span a wide range of different CDN request patterns. On average (across all cache sizes), LRB reduces WAN traffic by over 13% compared to B-LRU. Additionally, LRB is robust across all traces whereas no prior caching algorithm consistently improves the performance across traces and cache sizes. For example, LRU4 is the best on Wikipedia, but among the worst on other traces. LFUDA does well on Wikipedia, CDN-A1, but poorly on CDB-B3. TinyLFU does well on CDN-B3 and CDN-A2, but not on Wikipedia. These results indicate that heuristic-based algorithms work well with certain patterns and poorly with others.

To further understand where LRB's improvement comes from, Figure 10 shows the good decision ratio of LRB, the three best-performing state-of-the-art algorithms, and LRU. TinyLFU is not included as its implementation does not allow us to evaluate individual decisions. LRB achieves 74–86% good decision ratios, which are the highest for all but one of six combinations. This implies that LRB learns a better workload representation and is able to leverage its model to make good eviction decisions. Overall, we find that the good decision ratio of an algorithm strongly correlates with its byte miss ratio. One exception is CDN-B1 512 GB, where LeCaR has a higher good decision ratio than LRB, but with a worse byte miss ratio.

6.5 Comparison to Belady

We have seen that LRB provides significant improvements over state-of-the-art algorithms. We now compare LRB with the offline Belady MIN (oracle) and relaxed Belady algorithms. Figure 11 compares their byte miss ratios on three cache sizes. We further compare to the state-of-the-art policy, i.e., best performing policy, on each trace and cache size.

We find that LRB indeed reduces the gap between state-of-the-art algorithms and Belady, e.g., by about a quarter on most traces. While a significant gap still remains to Belady, LRB imitates relaxed Belady, which is thus a better reference point for LRB (Section 2). Relaxed Belady represents an ideal LRB with 100% prediction accuracy. The figure shows that LRB is closer to relaxed Belady, e.g., one third to half the distance on most traces. The remaining gap between LRB and relaxed Belady is due to our model's prediction error. This suggests that improvements in the prediction model are a promising direction for future work.



(a) Wikipedia (b) CDN-B1

Figure 12: Traffic reductions of LRB whose sliding memory window parameter are trained by a small portion of trace vs. LRB-OPT whose parameters are trained by the full trace.

6.6 Sliding Memory Window Size Selection

LRB tunes its memory window on a 20% validation prefix. However, as described in Section 4, the validation prefix is short compared to the optimal sliding memory window choice. To evaluate how much better LRB can do without this restriction, we experimentally determine the best sliding memory window for the full trace. We call the resulting LRB variant "LRB-OPT". LRB-OPT further improves the performance of LRB. Figure 12 shows the traffic reduction over B-LRU for the Wikipedia and CDN-B3 trace. For large cache sizes, LRB-OPT achieves an additional 1–4% traffic reduction compared to LRB. All in all, the ready availability of large amounts of training data in production will be highly beneficial for LRB.

7 Related Work

Although cache design has been studied since 1965 [84], we are only aware of three prior works that have studied how to leverage Belady's MIN [16]. Shepherd Cache [71] used a limited window to the future to emulate Belady's algorithm and defers replacement decisions until reuses occur. Hawkeye and Harmony [47, 48] applied Belady's algorithm to the history of memory accesses to help make better eviction or prefetch decisions. These previous efforts are for hardware caches (in the processor) and do not apply to software cache designs, especially with variable-sized objects.

The rest of the extensive literature can be divided into two major lines of work: caching algorithms and methods to adapt these algorithms to the workload.

Heuristic caching algorithms. We classify these algorithms by the features they use in the eviction decisions (Section 2). The two most widely used features are recency and frequency. Recency is typically measured as variants [3,44,52,68,70] of Delta_1 (as defined in Section 4). Some works also consider static features such as object sizes [4,5,12,24,37,76]. Application ids are commonly considered in shared caches [27,28]. Some algorithms rely entirely on frequency-based features (similar to individual EDCs) [7,11,38,53,63,75].

Another common approach is to combine a recency feature, a frequency feature, and an object's size. This is achieved with either a fixed weighting function between these features [5, 15, 18, 21, 26, 35, 40, 46, 56, 59, 69, 73, 86, 88], or by using a single-parameter adaptation method to dynamically set the weights [13, 19, 34, 49–51, 65, 81].

Two recent proposals consider a larger range of features [17, 55] and are evaluated using simulations. Unfortunately, they are either outperformed [55] or barely match [17] the miss ratio of recency/frequency-heuristics such as GDSF [26].

LRB uses a superset of all these features and introduces a low-overhead implementation in the ATS production system.

Workload adaptation methods. There are three different methods to adapt caching systems and their parameters to changing workloads.

Only a few papers propose to use machine learning as an adaptation method [8, 17, 20, 31, 32, 55, 87]. There are two branches. The first branch uses reinforcement learning [31,32,55,87], where the state-of-the-art is represented by UCB in our evaluation. The second branch uses supervised learning [17,20], where the state-of-the-art is represented by LFO in our evaluation. All of these proposals are evaluated using simulations. Unfortunately, both UCB and LFO perform much worse than simpler heuristics such as GDSF [26] on CDN traces (Section 6). In fact, recent work concludes that caching "is not amenable to training good policies" [55]. LRB overcomes this challenge using a different feature set (Section 4) and shows the feasibility of implementing machine learning in CDN production systems (Section 5).

The most common prior tuning approaches rely on shadow cache and hill climbing algorithms [13,19,28,34,50,52,56,65, 81, 82, 88]. Shadow caches are simulations that evaluate the miss ratio of a few parameter choices in parallel. Hill climbing repeatedly reconfigures the system to use the best parameter seen in these simulations, and restarts the simulations in the neighborhood of the new parameter.

Several caching systems rely on mathematical prediction models [9, 45, 64, 83, 85]. All heuristics-based algorithms and workload adaptation methods share the same fundamental limitation: they work well with certain workloads and poorly with others.

8 Conclusions

In this paper, we have presented the design, implementation, and evaluation of LRB and have shown that it is possible to design a practical ML-based CDN cache that approximates Belady's MIN algorithm and outperforms state-of-the-art approaches over 6 production CDN traces.

The key advantage of using ML to approximate Belady's MIN algorithm over access-pattern specific or heuristics-based approaches is that it can intelligently make cache eviction decisions based on any access pattern.

More importantly, we have introduced the relaxed Belady algorithm, Belady boundary, and good decision ratio as an eviction quality metric, which has enabled us to take a fundamentally new approach to caching that approximates Belady's MIN algorithm. We expect that these concepts can benefit others in the future.

We have shown that LRB's implementation is practical and deployable by replacing the caching component of a production CDN caching system with LRB. Our experiments show that its throughput and latency are on par with the native implementation. LRB requires neither GPUs nor accelerators and, in fact, its additional CPU and memory overheads are moderate and within the constraints of today's CDN server hardware. This deployable design is enabled by key design decisions including our feature selection, sliding memory window, training data selection, and choice of a lightweight machine learning model.

We have also shown that there is a sizable gap between LRB and the relaxed Belady offline algorithm. A promising direction of future work is to further improve the prediction accuracy of ML in LRB.

While we show tuning LRB's hyperparameters can be done on a small validation trace, we plan to further simplify deployment by automating the tuning of the sliding memory window parameter. These improvements will be available from LRB's repository at `https://github.com/sunnyszy/lrb`.

Acknowledgements

This research is partly funded by a Comcast Innovation grant, a Princeton University fellowship, and a Facebook research grant. We are grateful to our anonymous reviewers, our shepherd Rebecca Isaacs, Amit Levy, and David Liu whose extensive comments substantially improved this work. We also thank Fei Gao, Qizhe Cai, Rong Huang, Jeffrey Helt, and Jennifer Lam who contributed to the project at different stages.

References

[1] Companies using apache traffic server. https://trafficserver.apache.org/users.html. Accessed: 2019-04-22.

[2] Netlify. https://www.netlify.com/open-source/. Accessed: 2019-04-22.

[3] *A paging experiment with the multics system*. MIT Press, 1969.

[4] Marc Abrams, C. R. Standridge, Ghaleb Abdulla, S. Williams, and Edward A. Fox. Caching proxies: Limitations and potentials. Technical report, Virginia Polytechnic Institute & State University Blacksburgh, VA, 1995.

[5] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Edward A Fox, and Stephen Williams. Removal policies in network caches for World-Wide Web documents. In *ACM SIGCOMM*, pages 293–305, 1996.

[6] Micah Adler, Ramesh K Sitaraman, and Harish Venkataramani. Algorithms for optimizing the bandwidth cost of content delivery. *Computer Networks*, 55(18):4007–4020, 2011.

[7] Charu Aggarwal, Joel L Wolf, and Philip S Yu. Caching on the world wide web. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):94–107, 1999.

[8] Zahaib Akhtar, Yaguang Li, Ramesh Govindan, Emir Halepovic, Shuai Hao, Yan Liu, and Subhabrata Sen. Avic: a cache for adaptive bitrate video. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 305–317, 2019.

[9] George Almasi, Calin Cascaval, and David A Padua. Calculating stack distances efficiently. In *MSP/ISMM*, pages 37–43, 2002.

[10] Apache. Traffic Server, 2019. Available at https://trafficserver.apache.org/, accessed 09/18/19.

[11] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review*, 27(4):3–11, 2000.

[12] Hyokyung Bahn, Kern Koh, Sam H Noh, and SM Lyul. Efficient replacement of nonuniform objects in web caches. *IEEE Computer*, 35(6):65–73, 2002.

[13] Sorav Bansal and Dharmendra S Modha. CAR: Clock with adaptive replacement. In *USENIX FAST*, volume 4, pages 187–200, 2004.

[14] Novella Bartolini, Emiliano Casalicchio, and Salvatore Tucci. A walk through content delivery networks. In *IEEE MASCOTS*, pages 1–25, 2003.

[15] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *USENIX NSDI*, pages 389–403, 2018.

[16] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[17] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *ACM HotNets*, pages 134–140, 2018.

[18] Daniel S Berger, Sebastian Henningsen, Florin Ciucu, and Jens B Schmitt. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review*, 43(2):57–59, 2015.

[19] Daniel S. Berger, Ramesh Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *USENIX NSDI*, pages 483–498, 2017.

[20] Adit Bhardwaj and Vaishnav Janardhan. PeCC: Prediction-error Correcting Cache. In *Workshop on ML for Systems at NeurIPS*, December 2018.

[21] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC*, pages 499–511, 2017.

[22] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[23] Christopher J.C. Burges. From RankNet to LambdaRank to LambdaMART: An Overview. Technical report, Microsoft Research Technical Report MSR-TR-2010-82, 2010.

[24] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *USENIX symposium on Internet technologies and systems*, volume 12, pages 193–206, 1997.

[25] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. *ACM SIGCOMM*, 45(4):167–181, 2015.

[26] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *High-Performance Computing and Networking*, pages 114–123, 2001.

[27] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: dynamic cloud caching. In *USENIX HotCloud*, 2015.

[28] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*, pages 379–392, 2016.

[29] CISCO. Cisco visual networking index: Forecast and trends 2022, February 2019. Available at `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.pdf`, accessed 09/18/19.

[30] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *ACM SOSP*, pages 153–167, 2017.

[31] Renato Costa and Jose Pazos. Mlcache: A multi-armed bandit policy for an operating system page cache. Technical report, University of British Columbia, 2017.

[32] Jeff Dean. Is google using reinforcement learning to improve caching? Personal communication on 2018-09-27, September 2018.

[33] John Dilley, Bruce M. Maggs, Jay Parikh, Harald Prokop, Ramesh K. Sitaraman, and William E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[34] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *ACM Middleware*, pages 94–106, 2018.

[35] Gil Einziger and Roy Friedman. Tinylfu: A highly efficient cache admission policy. In *IEEE Euromicro PDP*, pages 146–153, 2014.

[36] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *USENIX NSDI*, pages 65–78, 2019.

[37] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI*, pages 371–384, 2013.

[38] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *ACM SoCC*, page 23, 2011.

[39] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[40] Nicolas Gast and Benny Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *ACM SIGMETRICS*, pages 123–136, 2015.

[41] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K Sitaraman. Trade-offs in optimizing the cache deployments of cdns. In *IEEE INFOCOM*, pages 460–468, 2014.

[42] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9, 2014.

[43] Leif Hedstrom. Deploying apache traffic server, 2011. Oscon.

[44] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *USENIX ATC*, pages 57–69, 2015.

[45] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. Fast miss ratio curve modeling for storage cache. *ACM Transactions on Storage (TOS)*, 14(2):12, 2018.

[46] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *ACM SOSP*, pages 167–181, 2013.

[47] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *ACM/IEEE ISCA*, pages 78–89, 2016.

[48] Akanksha Jain and Calvin Lin. Rethinking belady's algorithm to accommodate prefetching. In *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pages 110–123, 2018.

[49] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the clock replacement. In *USENIX ATC*, pages 323–336, 2005.

[50] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS*, 30(1):31–42, 2002.

[51] Shudong Jin and Azer Bestavros. GreedyDual* web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24:174–183, 2001.

[52] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[53] George Karakostas and Dimitrios N Serpanos. Exploitation of different types of locality for web caches. In *IEEE ISCC*, pages 207–212, 2002.

[54] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[55] Mathias Lecuyer, Joshua Lockerman, Lamont Nelson, Siddhartha Sen, Amit Sharma, and Aleksandrs Slivkins. Harvesting randomness to optimize distributed systems. In *ACM HotNets*, pages 178–184, 2017.

[56] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS*, volume 27, pages 134–143, 1999.

[57] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, (12):1352–1361, 2001.

[58] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage (TOS)*, 13(3):24, 2017.

[59] Cong Li. Dlirs: Improving low inter-reference recency set cache replacement policy with dynamics. In *ACM MSST*, pages 59–64, 2018.

[60] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *EUROSYS*, pages 1–15, 2015.

[61] Ping Li. Robust logitboost and adaptive base class (abc) logitboost. *arXiv preprint arXiv:1203.3491*, 2012.

[62] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. Model ensemble for click prediction in bing search ads. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 689–698, 2017.

[63] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM CCR*, 45:52–66, 2015.

[64] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[65] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *USENIX FAST*, volume 3, pages 115–130, 2003.

[66] Kianoosh Mokhtarian and Hans-Arno Jacobsen. Caching in video cdns: Building strong lines of defense. In *ACM EuroSys*, page 13, 2014.

[67] Devon H. O'Dell. Personal communication at Fastly.

[68] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD*, 22(2):297–306, 1993.

[69] Sejin Park and Chanik Park. Frd: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *ACM MSST*, pages 59–64, 2017.

[70] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jinsoo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *ACM/IEEE CASES*, pages 234–241, 2006.

[71] Kaushik Rajan and Govindarajan Ramaswamy. Emulating optimal replacement with a shepherd cache. In *IEEE/ACM MICRO*, pages 445–454, 2007.

[72] Matthew Richardson, Ewa Dominowska, and Robert Ragno. Predicting clicks: estimating the click-through rate for new ads. In *Proceedings of the 16th international conference on World Wide Web*, pages 521–530, 2007.

[73] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. *IEEE/ACM TON*, 8:158–170, 2000.

[74] Emanuele Rocca. Running Wikipedia.org, June 2016. Available at `https://www.mediawiki.org/wiki/File:WMF_Traffic_Varnishcon_2016.pdf`, accessed 09/18/19.

[75] Ketan Shah, Anirban Mitra, and Dhruv Matani. An O(1) algorithm for implementing the LFU cache eviction scheme. Technical report, Stony Brook University, 2010.

[76] David Starobinski and David Tse. Probabilistic methods for web caching. *Performance evaluation*, 46:125–137, 2001.

[77] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *ACM CoNEXT*, pages 55–67, 2017.

[78] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: advanced photo caching on flash for facebook. In *USENIX FAST*, pages 373–386, 2015.

[79] Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. Popularity prediction of facebook videos for higher quality streaming. In *USENIX ATC*, pages 111–123, 2017.

[80] Ilya Trofimov, Anna Kornetova, and Valery Topinskiy. Using boosted trees for click-through rate prediction for sponsored search. In *Proceedings of the Sixth International Workshop on Data Mining for Online Advertising and Internet Economy*, pages 1–6, 2012.

[81] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *USENIX HotStorage*, 2018.

[82] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *USENIX ATC*, pages 487–498, 2017.

[83] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *USENIX FAST*, pages 95–110, 2015.

[84] Maurice V Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions Electronic Computers*, 14(2):270–271, 1965.

[85] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *USENIX OSDI*, pages 335–349, 2014.

[86] Roland P Wooster and Marc Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 29(8):977–986, 1997.

[87] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. A deep reinforcement learning-based framework for content caching. In *IEEE CISS*, pages 1–6, 2018.

[88] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX ATC*, pages 91–104, 2001.

Meaningful Availability

Tamás Hauer *Google*   Philipp Hoffmann *Google*   John Lunney *Google*   Dan Ardelean *Google*   Amer Diwan *Google*

Abstract

High availability is a critical requirement for cloud applications: if a sytem does not have high availability, users cannot count on it for their critical work. Having a metric that meaningfully captures availability is useful for both users and system developers. It informs users what they should expect of the availability of the application. It informs developers what they should focus on to improve user-experienced availability. This paper presents and evaluates, in the context of Google's G Suite, a novel availability metric: windowed user-uptime. This metric has two main components. First, it directly models user-perceived availability and avoids the bias in commonly used availability metrics. Second, by simultaneously calculating the availability metric over many windows it can readily distinguish between many short periods of unavailability and fewer but longer periods of unavailability.

1 Introduction

Users rely on productivity suites and tools, such as G Suite, Office 365, or Slack, to get their work done. Lack of *availability* in these suites comes at a cost: lost productivity, lost revenue and negative press for both the service provider and the user [1, 3, 6]. System developers and maintainers use metrics to quantify service reliability [10, 11]. A good availability metric should be *meaningful*, *proportional*, and *actionable*. By "meaningful" we mean that it should capture what users experience. By "proportional" we mean that a change in the metric should be proportional to the change in user-perceived availability. By "actionable" we mean that the metric should give system owners insight into why availability for a period was low. This paper shows that none of the commonly used metrics satisfy these requirements and presents a new metric, *windowed user-uptime* that meets these requirements. We evaluate the metric in the context of Google's G Suite products, such as Google Drive and Gmail.

The two most commonly used approaches for quantifying availability are *success-ratio* and *incident-ratio*. Success-ratio is the fraction of the number of successful requests to total requests over a period of time (usually a month or a quarter) [5, 2, 9]. This metric has some important shortcomings. First, it is biased towards the most active users; G Suite's most active users are 1000x more active than its least active (yet still active) users. Second, it assumes that user behavior does not change during an outage, although it often does: e.g., a user may give up and stop submitting requests during an outage which can make the measured impact appear smaller than it actually is. Incident-ratio is the ratio of "up minutes" to "total minutes", and it determines "up minutes" based on the duration of known incidents. This metric is inappropriate for large-scale distributed systems since they are almost never completely down or up.

Our approach, *windowed user-uptime* has two components. First, user-uptime analyzes fine-grained user request logs to determine the up and down minutes for each user and aggregates these into an overall metric. By considering the failures that our users experience and weighing each user equally, this metric is meaningful and proportional. Second, windowed user-uptime simultaneously quantifies availability at all time windows, which enables us to distinguish many short outages from fewer longer ones; thus it enables our metric to be actionable.

We evaluate windowed user-uptime in the context of Google's G Suite applications and compare it to success-ratio. We show, using data from a production deployment of G Suite, that the above-mentioned bias is a real shortcoming of success-ratio and that windowing is an invaluable tool for identifying brief, but significant outages. Our teams systematically track down the root cause of these brief outages and address them before they trigger larger incidents.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 motivates the need for *windowed user-uptime* Section 4 describes user-uptime. Section 5 extends user-uptime with windowed user-uptime. Section 6 evaluates our approach and Section 8 concludes the paper.

2 Related Work

Understanding and improving availability of computer systems has been a goal for decades [22] with early work aiming at developing stochastic models based on failure characteristics of individual hardware and software components [30, 20, 27, 21] . A more abstract approach [28] uses mean time to failure (MTTF) and mean time to recovery (MTTR), which are well suited to describe a system that shifts between binary up/down states. An extension to this approach that differentiates between transient failures and short-term downtime can be found in Shao *et al.* [26].

When describing complex distributed systems, however, a binary up/down state is usually not accurate. Brown *et al.* [13] focus on success-ratio instead, describing the availability of a system as the percentage of requests served successfully and understanding it as a spectrum between up and down.

Brevik *et al.*[12] show how to compute confidence bounds for future performance based on collected past performance data. Treynor *et al.* [29] describe error budgets connected to an availability goal and how availability of individual components affects the whole system.

In taking a user-centric approach, Dell'Amico *et al.*[18] explore problems similar to those presented here, but with a focus on prediction and future performance. In a sequel[17], a probabilistic model is used to suggest optimizations to a network application.

A common application for availability and error budgets are contracts for cloud service providers [5, 2, 9]. Patel *et al.* [25] discuss these Service Level Agreements (SLAs) focusing on a framework to define and monitor metrics and goals associated with SLAs while Endo *et al.* [19] describe issues encountered when trying to achieve high availability in cloud computing.

Microsoft Cloud services (such as Office 365) compute and report uptime as the time when the overall service is up divided by total time. [4] As a metric based on time, it is immediately meaningful (e.g., everyone interprets 99.9% uptime as "system will be down for a total of 1 day every 1000 days on average").

Before the work in this paper, Gmail computed and reported availability as the percentage of successful interactive user requests (irrespective of which user the request comes from). Google Cloud Platform computes and reports "downtime" as the error rate and downtime period as consecutive minutes of downtime.

Slack's status page reports that it is "a distributed platform and during any given incident it is rare for all Slack teams to be affected. For this reason, we report our uptime as an average derived from the number of affected users." [7]

Amazon Web Services computes and reports "Error rate" as the percentage of requests that result in errors in a 5 minute interval. The average of these five minute calculations is reported to customers, as "Monthly uptime percentage" - the average of all the 5-minute error rates. [2]

In the following sections, we will look at three desirable properties of an availability metric: meaningful, proportional and actionable. The surveyed systems each satisfy some but not all of these three properties.

3 Motivation

Availability is the ability of a service to perform its required function at an agreed instant or over an agreed period of time [24]. At a high level, all availability metrics have the following form:

$$\text{availability} = \frac{\text{good service}}{\text{total demanded service}} \tag{1}$$

Availability metrics are invaluable to both users and developers [10].

For users, they tell them whether or not a service is suitable for their use case. For some use cases, unavailability translates into lost productivity and a *meaningful* measure can help quantify that.

For developers, availability metrics help prioritize their work to improve their system. For this to be the case, the measure should be *proportional* and *actionable*. A proportional metric enables developers to quantify the benefit of an incremental improvement. An actionable metric enables them to zero in on episodes of worst availability and thus find problems that they need to address.

Broadly speaking, availability metrics fall in two categories: *time based* and *count based*, according to how they quantify the "good service" and "total demanded service" in Eq. (1).

3.1 Time-based availability metrics

Toeroe *et al.* [28] define availability as $\frac{\text{MTTF}}{\text{MTTF+MTTR}}$ where MTTF is the mean-time-to-failure and MTTR is mean-time-to-recovery. This measure is based on the duration when the system is up or down; these concepts are meaningful to users. Thus it is no surprise that many cloud providers, e.g. Microsoft's Office 365 [4], use it. The time between failures is uptime and the time to recover from a failure is downtime, thus we can express this metric as Eq. (1) once we identify uptime and downtime as the measure of good and bad service, respectively [29]:

$$\text{availability} = \frac{\text{uptime}}{\text{uptime} + \text{downtime}}. \tag{2}$$

At a deeper level, this measure depends in a complex way on the precise meanings of "failure" and "recovery". At one extreme we could consider a system to have a failure only when there is an outage that affects *all* users. At the other extreme, we could consider a system to have a failure when there is an outage that affects even a *single* user.

Neither of these definitions are adequate. The first is unsatisfactory because the design and deployment of cloud systems, such as Azure [14], Dynamo [16], or Gmail, actively avoid single points of failure by sharding data across many machines and using replication [23] with failover when problems occur. Consequently, these systems rarely have an outage that affects all users [29]. The other extreme is not suitable, particularly for systems with hundreds of millions of users, because there will always be some that are experiencing failures. The system may not even be at fault for such outages. For example, we once debugged a situation where a user had authorized access to their Gmail account to many third-party services, each of which was repeatedly making extremely large requests against the user's account, inadvertently and collectively causing a denial of service attack, and in turn an outage for that user. It would be unreasonable to consider this as an outage of Gmail.

Consequently, availability in terms of up and down states either require manual labeling of up and down times (e.g., the incident ratio metric) or the use of thresholds, e.g., the G Suite SLA defines downtime as "if there is more than a five percent user error rate" [5]. While such thresholds avoid the extremes above, they do so arbitrarily. Furthermore, availability metrics that use such thresholds are not proportional. For example, the G Suite definition treats a system with 5% error rate the same as a system with 0.0001% error rate; and it treats a system with 5.1% error rate the same as a system with 99% error rate.

Thus, commonly used time-based availability metrics:

- are not proportional to the severity of the system's unavailability (a downtime with 100% failure rate weighs as much as one with 10%).
- are not proportional to the number of affected users (a downtime at night has the same weight as a downtime during peak period).
- are not actionable because they do not, in themselves, provide developers guidance into the source of failures.
- are not meaningful in that they rely on arbitrary thresholds or manual judgments (e.g., incident-ratio).

3.2 Count-based availability metrics

While the most common definitions of availability (including the ones discussed above) are in terms of time, some systems use "success-ratio" instead. Success-ratio is the ratio of successful requests to total requests. Since success-ratio does not use a threshold, it is more proportional than commonly-used time-based metrics.

Success-ratio as an availability measure is popular because it is easy to implement and is a reasonable measure: it accurately characterizes a service, whose (un-)reliability stems solely from a stochastic distribution of failures. Because the computation is based on user requests, it approximates user perception better than if we used some internal instrumentation to reason about the service's health. It is, however, prone to bias: the most active users are often 1000x more active than the least active users and thus are 1000x over-represented in the metric.

Even if we disregard the bias with respect to different users' activity, this metric can fail to proportionally capture changes in availability. Consider, for example, that a service is down for 3 hours and then up for 3 hours. While the system is up, users are active on the system and thus make many (successful) requests to the system. While the system is down, they will periodically probe the system to check if is still down but the inter-arrival time of the requests is likely lower than if the system is in active use. For certain systems, particularly ones that provide a service to other systems, the opposite situation may hold: a client service may flood our system with retries thus inflating failure count. Users ultimately care about time, thus a count-based measure - in contrast to a time-based one - can misrepresent the magnitude of an outage.

In summary, count-based (success-ratio) availability metrics:

- are not meaningful in that they are not based on time.
- are biased by highly active users.
- are biased because of different client behavior during outages.

3.3 Probes

Using synthetic probes may mitigate some of the shortcomings of success-ratio. For example, probing the system automatically at regular intervals can avoid bias. This approach works well for low-level systems with modest business logic, for example, a network which "just" sends and receives packets. In contrast, a cloud application may have hundreds of types of operations and the work required by an operation type depends on the type and the user's corpus (e.g., a search operation for a user with a few documents is fundamentally different, in terms of the work involved, from a search for a user with millions of documents). Consequently, despite years of effort, we have been unable to fully represent real workloads on our system with synthetic probes [8]. Thus, for cloud applications, a metric which uses synthetic probes may not be representative of real-user experience.

In summary, availability metrics based on synthetic probes:

- are not representative of user activity.
- are not proportional to what users experience.

3.4 Actionable metrics

The availability metrics that we have described so far represent different points in the tradeoffs for "proportional" and "meaningful". All of them, however, have a similar weakness when it comes to being "actionable": a single number associated with a reporting period does not allow enough insight into the source or the shape of unavailability.

Indeed, none of the existing metrics can distinguish between 10 seconds of poor availability at the top of every hour or 6 hours of poor availability during peak usage time once a quarter. The first, while annoying, is a relatively minor nuisance because while it causes user-visible failures, users (or their clients) can retry to get a successful response. In contrast the second is a major outage that prevents users from getting work done for nearly a full day every quarter.

In the following section, we describe a new availability measure, user-uptime that is meaningful and proportional. Afterwards, we'll introduce windowed user-uptime, which augments it to be actionable.

4 Proportional and meaningful availability: user-uptime

As discussed in Section 3, prior metrics for availability are not meaningful or proportional. A seemingly straightforward variant of Eq. (2) satisfies both proportionality and meaningfulness:

$$\text{user-uptime} = \frac{\sum_{u \in \text{users}} \text{uptime}(u)}{\sum_{u \in \text{users}} \text{uptime}(u) + \text{downtime}(u)}, \qquad (3)$$

Since this metric is in terms of time, it is meaningful to users. Since it is free from arbitrary thresholds, it is (inversely) proportional to the duration and magnitude of the unavailability. The calculation of this metric, however, is not straightforward as it requires a definition of uptime(u) and downtime(u) *per user*. The remainder of this section describes how we calculate these.

4.1 Events and durations

An obvious approach to uptime and downtime would be to introduce evenly spaced synthetic probes for *each* user (Figure 1) and count the successful and failing probes. In Figure 1 the light green circles are successful events and dark red diamonds are failed ones from a single user's perspective. The green horizontal line at the top marks the period when the user perceives the system to be up (i.e., its length represents uptime) while the red one on the bottom marks the period when the user perceives the system to be down. As discussed in Section 3.1, however, synthetic probes fail to

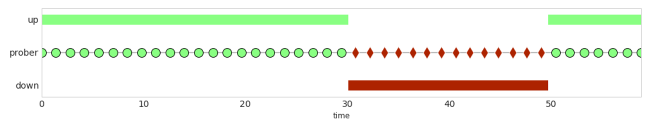


Figure 1: System availability as seen through evenly-spaced prober requests. success-ratio=67%, user-uptime=67%

mimic true user behavior. For example, we may probe certain operations but users may experience unavailability for other operations. In a recent Gmail outage, for example, the failure of the system serving attachments impacted only operations that accessed attachments; but other operations were largely unaffected.

Our key insight, therefore, is to use user requests as probes. A user's perception of a system being up or down depends on the response they get when they interact with it: if the response is successful (unsuccessful) they perceive the system as being up (down). For example, as long as the user can successfully edit a document, she perceives the system to be up; once she experiences a failing operation she perceives the system to be down.

Success versus failure of individual events is not always obvious to the user: for example, if a device's request to synchronize the contents of the user's mailbox fails, the user may not notice that the system is down. Despite this, we want our availability metric to be conservative: if there is *any* chance that a user may perceive a failure (even one as subtle as a longer-than-usual delay in the notification for a new email) we consider it as a failure.

Fig. 2 illustrates this approach. In contrast to Fig. 1, availability is not necessarily the same as success-ratio. Section 4.2 discusses some variations of this approach and the consequences of our choices.

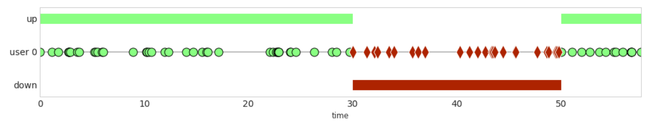


Figure 2: Availability as seen through unevenly spaced requests from a single user. success-ratio=68%, user-uptime=65%

Fig. 3 illustrates a system with four users. Each user experiences an outage of a different duration and generates requests at a different rate from the other users. Thus if we use success-ratio across the users, it will skew availability towards the most prolific user. User-uptime does not suffer from this bias.

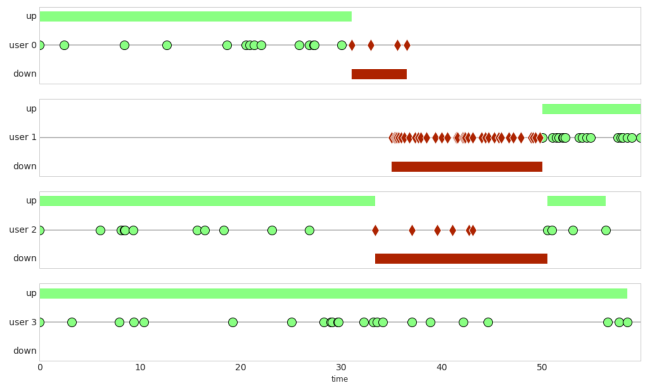


Figure 3: An outage which affects users selectively.

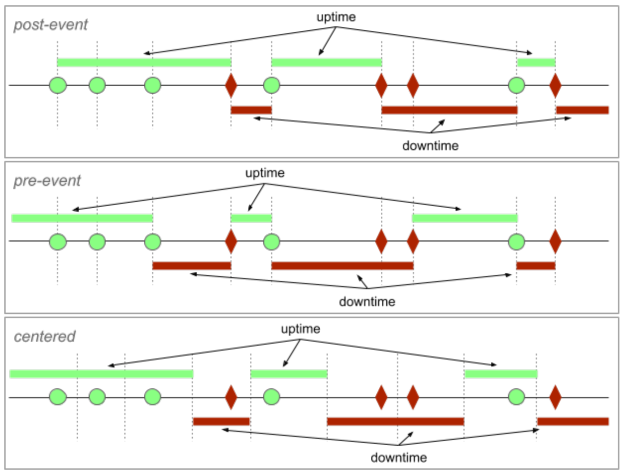


Figure 4: Three choices to extrapolate uptime or downtime from neighboring events

4.2 Challenges with user uptime

While we have illustrated user-uptime using examples, there are two main questions that we have glossed over: (i) how do we label a duration as up or down; and (ii) how do we address users that are inactive. The remainder of this section addresses these challenges.

4.2.1 Labeling durations

As long as back-to-back events are both failures or successes, it is obvious how to label the duration between the two events. But if back-to-back events are different (i.e., one is a failure and one is a success) there are three choices (Fig. 4):

- After a successful (or failing) operation, assume that the system is up (or down) until the user sees evidence to the contrary.
- Before a successful (failing) operation, assume that the system is up (down) until the previous event. Intuitively, this option takes the position that the user request probes the state (up or down) of the system up to the previous operation.
- Split the duration between events (i.e., half the time is uptime and half is downtime).

Assuming that transitions (from up to down or vice versa) occur at random points between the events, the differences between the above three options will be negligible. This is not always the case when the client aggressively retries failing operations. For simplicity, and because it captures user intuition of system availability, we use the first option.

4.2.2 Active and inactive periods

If a user stops all activity for an extended period (e.g., goes on a vacation) they have no perception of the system being up or down: assuming that the system is up or down continuously after the last seen request does not make sense and may even optimistically bias the data: e.g., if a user has an unsuccessful request they may retry until they get success and thus the last request before a vacation is disproportionately likely to be successful.

To this end, we introduce a *cutoff* duration; if a user has been inactive for more than this duration, we consider the user as being inactive on the system and thus do not count uptime or downtime. We pick the cutoff duration as the 99th percentile of the interarrival time of user requests. For Gmail this is 30 minutes. Experiments with multiples of this duration have shown that our availability metric does not change significantly with different values of the cutoff duration.

Now we can define how we label durations as uptime or downtime for each user:

Definition (uptime, downtime): A segment between two consecutive events originating from the same user is:

- *inactive* if the two events are further apart than `cutoff`, otherwise
- *uptime* if the first of the two events was *success*
- *downtime* if the first of the two events was *failure*.

Fig. 5 illustrates this definition.

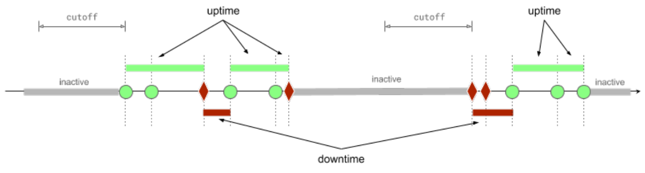


Figure 5: Definition of uptime and downtime segments

For each user u and a measurement period of interest, uptime(u) is the sum of the lengths of the uptime segments and downtime(u) is the sum of the lengths of the downtime segments. We can calculate overall availability for the system by aggregating across all users as in Eq. (3).

4.3 Properties of user-uptime

We now study the properties of user-uptime and success-ratio in synthetic situations where we know the ground truth; later (Section 6) we evaluate them in a production system.

For the first example, we generated synthetic user requests for an hour such that all user requests within a contiguous 15 minute period failed. We simulated this thousands of times for 10,000 synthetic active users. Fig. 6 illustrates what this outage looks like for 2 users.

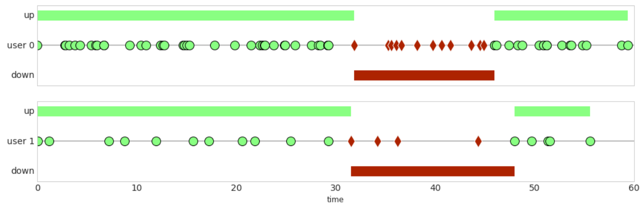


Figure 6: Two users' activity around a hard outage during $30 < t < 45$

Fig. 7 shows the distribution of the success-ratio and user-uptime metrics across the different simulations. We see that both metrics show an availability of around 0.75 but the standard deviation for user-uptime is much smaller indicating that user-uptime more precisely and consistently captures the outage.

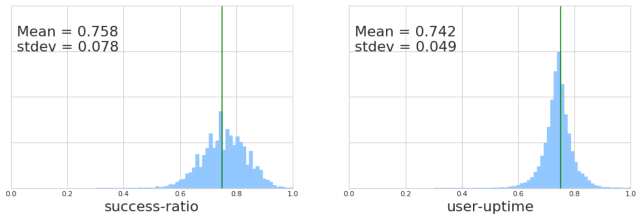


Figure 7: Normalized distribution of success-ratio and user-uptime measurements

For the second example, we incorporated retries: when a request fails, the user retries the request. Fig. 8 shows the distribution for success-ratio and user-uptime for this experiment. We see that user uptime is more accurate than success-ratio: identifying the availability of 0.75 with a tight distribution. Success-ratio, on the other hand, is affected by the retries and indicates a lower availability than what users actually experienced.

In summary, at least with synthetic examples we see that user-uptime better captures availability than success-ratio. Section 6 compares the two metrics using production data.

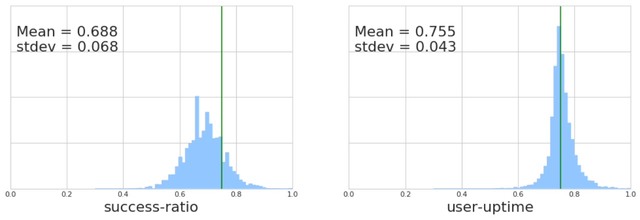


Figure 8: Normalized distribution of success-ratio and user-uptime measurements with automatic retries

5 Actionable availability: windowed user-uptime

Cloud productivity suites commonly define, track, and report monthly or quarterly availability. As Section 3.4 points out, monthly or quarterly availability data is often not actionable. Concretely, monthly availability does not distinguish between a flaky system that routinely fails a small percentage of the requests from a system whose failures are rare but when they happen the system is unavailable for an extended duration.

To distinguish long outages from flakiness, we must look at availability at the timescale of the outages: looking at a timescale of months may average away the unavailability and paints a rosier picture than reality. Our approach, *windowed user-uptime*, addresses this by combining information from all timescales simultaneously. The rest of this section describes windowed user-uptime and explores its properties.

5.1 Calculating windowed user-uptime

Windowed user-uptime iterates over all time windows fully included in the period of interest (e.g., month or quarter) and it computes an availability score for each window size. For example, to calculate windowed user-uptime over the month of January 2019, windowed user-uptime finds the worst period of each window size from 1 minute up to a month. The score for a window size w is the availability of the *worst* window of size w in our period of interest. We calculate the availability for a particular window from t_1 to t_2 as follows:

$$A(t_1, t_2) = \frac{\text{good service between } t_1 \text{ and } t_2}{\text{total service between } t_1 \text{ and } t_2}$$

To obtain the score for a window of size w, we enumerate all windows of duration w, compute the availability for each of them, and take the lowest value. Thus, we end up with one score for each window size. We call this score the minimal cumulative ratio (MCR).

Formally, the *MCR* for a window size w in a period (T_1, T_2) is as follows:

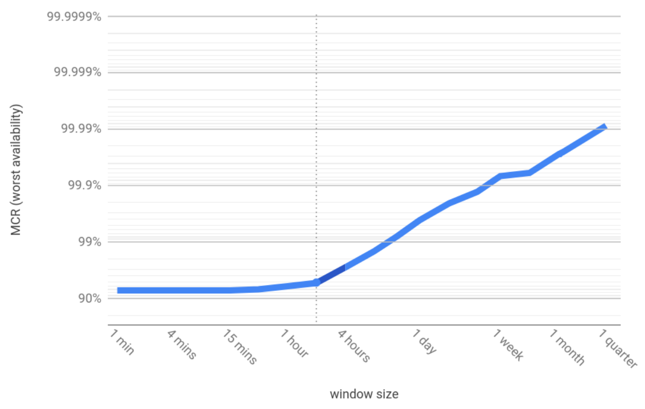


Figure 9: Windowed user-uptime:

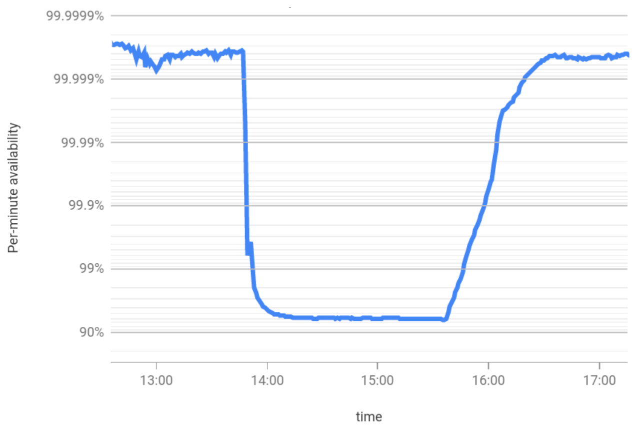


Figure 10: Per-minute availability over time

$$\mathrm{MCR}(w) \equiv \min_{T_1<t_1<t_2<T_2} \{A(t_1,t_2)|t_2-t_1=w\} \tag{4}$$

MCR picks the worst availability for each window size because that is the window that had the most impact on overall availability. Prior work in evaluating real-time garbage collectors and specifically the Minimum-Mutator-Utilization metric [15] inspired our windowing approach.

Fig. 9 shows an example of windowed user-uptime over a period of one quarter; the x-axis gives the window size and the y-axis (in log scale) gives the corresponding MCR. Fig. 9 enables us to readily make the following observations:

- The overall availability for the quarter is 99.991%; this is the rightmost point in the curve.

- There was no window of one minute or longer whose availability was worse than 92%.

- Knees of the curve can give insight into the longest incidents and their distribution for the service. In the example, the knee at about 2 hours indicates that the episode that brought availability down to 92% lasted for two hours; availability rapidly improves for larger windows.

By maintaining a mapping from each window size to the start of the window, we can readily examine the worst window(s) of a particular size in detail (e.g., Fig. 10). We can automatically discover the knee by finding the peak of the 2nd derivative of the windowed graph.

In summary, windowed user-uptime provides us with a rich view into the availability of our services. This data is actionable: it tells us the windows that degrade our overall availability. Our engineers use these graphs to find and fix sources of unavailability.

5.2 Monotonicity with integer-multiple sized windows

Intuitively, we expect windowed user-uptime to be monotonically non-decreasing in the size of the window; i.e., we expect larger windows to have better availability. This section proves that this is the case as long as window sizes are integer multiples of smaller ones.

We use boldface to distinguish windows: $\mathbf{w} = [t_1,t_2]$ from window sizes: $w = t_2 - t_1$. For the remainder of this section a window $\mathbf{w}$ will always be a continuous interval. The availability of a window $\mathbf{w}$ is the ratio of uptime and total time[1] over that window:

$$A(\mathbf{w}) = \frac{u(\mathbf{w})}{t(\mathbf{w})}.$$

Given a period of interest, $[T_1,T_2]$, windowed user-uptime, or the *minimal cumulative ratio* is the least of the availabilities of all windows (Eq. (4)) of *size* w that are fully enclosed within $[T_1,T_2]$:

$$\mathrm{MCR}(w) \equiv \min_{\substack{\mathbf{w}\subseteq[T_1,T_2]\\ |\mathbf{w}|=w}} \left(\frac{u(\mathbf{w})}{t(\mathbf{w})}\right).$$

One expects $\mathrm{MCR}(w)$ to be a monotonic function. Indeed, the availability over a window $\mathbf{w}'$ of size w' is intuitively the mean of a fluctuating time series. Scanning the interval with windows of smaller size $w < w'$, we should find both higher and lower availability values and the minimum of these should be smaller than the mean over the whole window:

$$\mathrm{MCR}(w) \overset{?}{\leq} MCR(w') \qquad w \leq w'. \tag{5}$$

[1]We use user-uptime terminology but windowing can be defined for other availability metrics. Substitute the corresponding concepts for good and total service, for example request count in case of success-ratio.

Eq. (5) indeed holds when we can cover a larger window fully using windows of the next smaller size without overlap. Consider $\mathbf{w}'$ of size w' covered with windows of size w as in Fig. 11

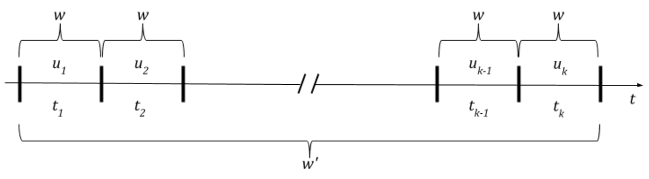


Figure 11: Covering a window with smaller ones: $w' = kw$

We denote the uptime and total time of the i^{th} window as u_i and t_i, respectively. Then the availability of $\mathbf{w}'$ can be written as:

$$A(\mathbf{w}') = \frac{u_1 + u_2 + \ldots + u_k}{t_1 + t_2 + \ldots + t_k}.$$

It is impossible that all of the ratios u_i/t_i are greater than $A(\mathbf{w}')$. Indeed, the above equation can be rearranged to

$$0 = \sum_{i=1}^{k} (u_i - A(\mathbf{w}')t_i)$$

and at least one term in the sum must be non-positive for the sum to yield zero: $u_j \leq A(\mathbf{w}')t_j$. Therefore there is at least one window of size w whose availability is at most that of $\mathbf{w}'$:

$$\frac{u_j}{t_j} \leq A(\mathbf{w}').$$

It follows that every window of length kw contains at least one window of size w with lower availability. Since there exists a window whose availability is MCR(kw), it follows that:

$$\mathrm{MCR}(w) \leq \mathrm{MCR}(kw) \qquad \forall k \in \mathbb{N} \tag{6}$$

5.3 Monotonicity in the general case

For practical purposes, Eq. (6) is “enough”: the worst availability of a day is always better than the worst availability of an hour or of a minute. Curiously, windowed user-uptime is not monotonically non-decreasing in the general case. The Appendix gives a proof that we can bound the extent of this apparent anomaly:

$$\frac{k}{k+1}\mathrm{MCR}(w) \leq \mathrm{MCR}(w') \qquad kw < w'. \tag{7}$$

In practice, we can guard against the apparent anomaly and enforce strict monotonicity by restricting to integer multiples, e.g. compute windowed user-uptime for windows that are powers of two.

6 Evaluation

Since last year, we have used windowed user-uptime for all G Suite applications (Calendar, Docs, Drive, Gmail, etc.). This section presents case studies where this metric provided keen insight into availability especially compared to the success-ratio metric.

We present availability data from production Google servers; in doing so we ignore unavailability due to issues with user devices or mobile networks.

Since the actual availability data is Google proprietary, we linearly scale and shift all curves in our graphs. This transformation preserves the shape of the curves and enables meaningful comparison between them.

6.1 Availability due to hyper-active users

As we have discussed, bias (due to hyper-active users) can negatively affect success-ratio. This section shows a real-life example of this phenomenon.

Fig. 12 shows (scaled) user-uptime and success-ratio for a large user base. One observes a consistent and significant mismatch between success-ratio and user-uptime.

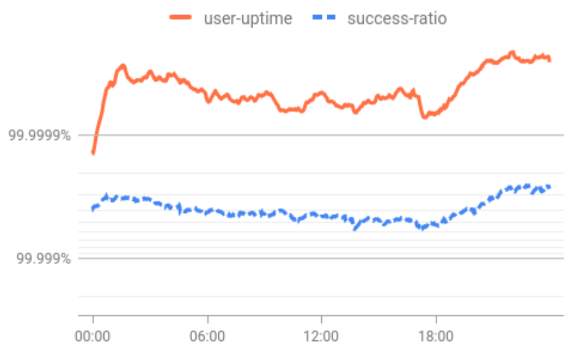


Figure 12: Uptime discrepancy due to mixed use-cases (log scale)

Upon diving into this discrepancy, we discovered that most (99%) active users contribute fewer than 100 events each in the duration of the graph. The remaining (1%) of the users are hyper-active and contribute 62% of all events.

Fig. 13 breaks down the availability data. The “Overall” curves give overall availability, the “Hyperactive users” curves give availability for the most active 1% of the users, and the “Typical users” curves give availability for the rest of them. Success-ratio appears heavily biased towards the availability for hyper-active users even though they make up only 1% of the user base.

Bias due to hyper-active users for success-ratio is, thus, not a theoretical possibility: we see this in production: indeed our most active users are 1000x more active than our

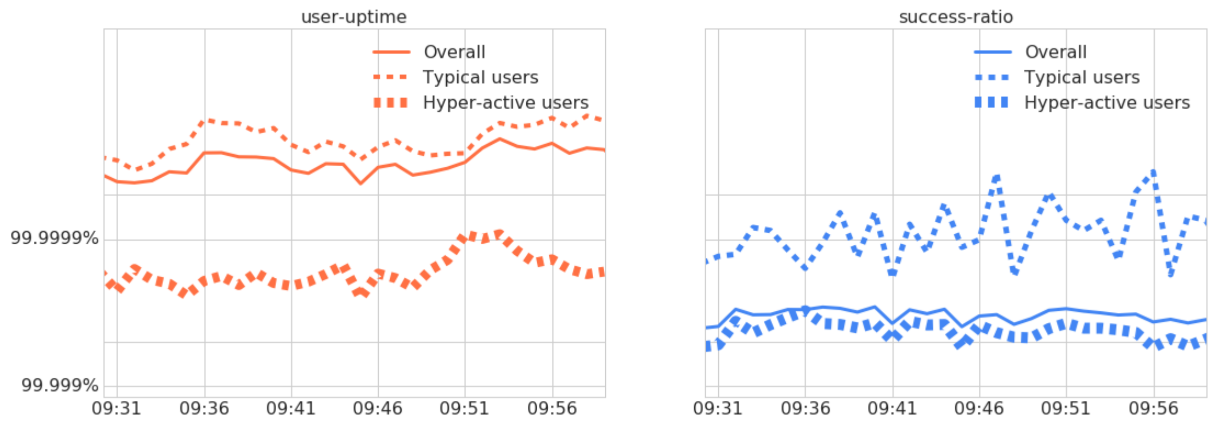


Figure 13: Left: user-uptime for the two clusters of events and the combined user-uptime. Right: success-ratio for the two clusters of events and the combined success-ratio

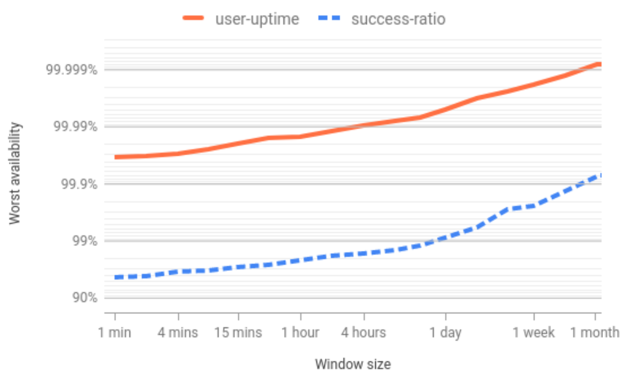


Figure 14: Monthly windowed uptime: success-ratio and user-uptime

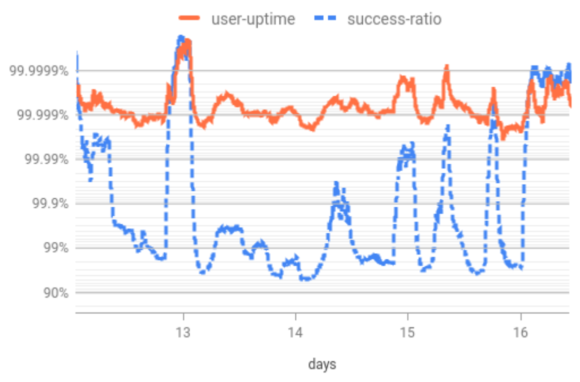


Figure 15: Effect of abusive users

"typical" users which makes this bias a common occurrence. When this bias occurs, success-ratio can mislead us towards thinking that an incident is much more or much less severe than it actually is.

6.2 Availability and hyper-active retries

While so far we have shown availability data for all customers of a G Suite application, we also separately measure availability for large customers since they often have their own unique patterns of usage and thus may experience different availability from other users. From this data we noticed that a particular large customer ($> 100,000$ users) had much poorer success-ratio than other users of Gmail; moreover we noticed a discrepancy between success-ratio and user uptime (Fig. 14). While the curves for user-uptime and success-ratio have the same shape, they are far apart with user-uptime indicating a better availability than success-ratio.

Fig. 15 shows the user-uptime and success-ratio over time. We can see that before and after the incident (i.e., the two ends of the graph) the two metrics were similar, but during the incident the two metrics diverged.

On investigation we uncovered that a small number of users had enabled a third-party application which synced their mailboxes; this external service was unable to handle large mailboxes and for these users it would retry the sync operation repeatedly and without exponential back-off. This generated bursts of failing requests. The event count of these failures drove the success-ratio availability down, even though this impacted only a handful of users and other requests for the users (e.g., to read an email) were successful. User-uptime did not suffer from the bias due to these retries. We were able to resolve this incident by communicating with the third-party vendor.

In summary, users (and the clients they use) can behave differently during incidents than during normal operations. Clients may make many more requests during incidents (our example above) or they may just decide to give up on the

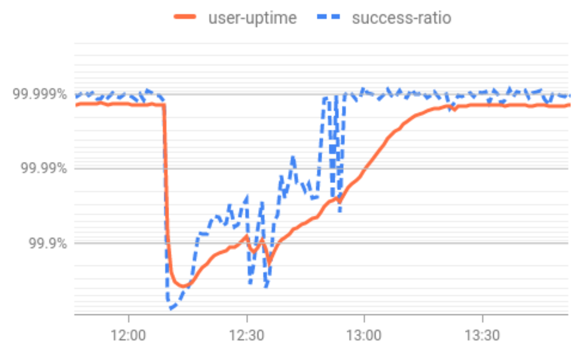


Figure 16: Impact assessment

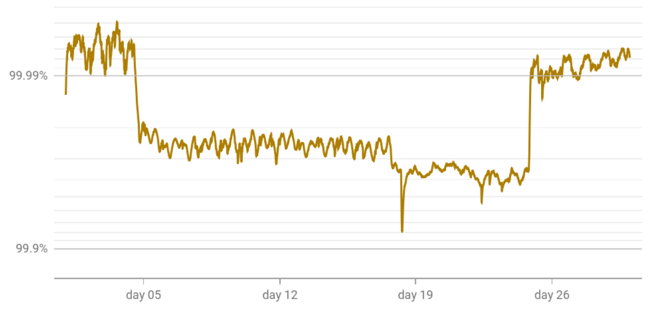


Figure 17: User-uptime analysis of Google Drive traffic, 30 days.

system and try few hours later. In both cases, success-ratio over- or under-estimates the magnitude of an outage while user-uptime matches user perception.

6.3 Quantifying impact of outages

We quantify the impact of every outage on our users. This impact advises the aftermath of the outage: for example, a really severe outage may result in a temporary freeze in deploying new versions until we address all the underlying issues while a less severe outage may require less extreme measures.

When using success-ratio, this quantification is difficult since the metric is not based on time. Concretely, since request patterns (and specifically retry behavior) changes during outages, using success-ratio to derive impact is not meaningful.

Consider, for example Fig. 16 which displays an outage from one of our products. Both success-ratio and user-uptime dip at 12:10 to around 97%, then recover over the course of an hour. What impact did this outage have on our users? Was it the case that $100\% - 97\% = 3\%$ of our users couldn't work for over an hour? Or that the impact was more evenly spread across our users and that automatic retries hid most of the negative impact?

The seconds of downtime, which we compute as part of user uptime, provides more insight. From there we see the minutes of downtime that our users experience. The large number we measured for this indicated that this was clearly a significant outage for some users and the retry behavior was unable to mask this outage from our users.

6.4 Combining user-uptime and success-ratio

The Drive team extensively uses windowed user-uptime to investigate, root-cause and fix sources of unavailability. The proportionality of user-uptime is critical for their use case: they need to be able to make changes and notice a proportional change in the availability metric. Sometimes, combining user-uptime with success-ratio yields valuable insights.

Fig. 17 illustrates this situation. A first analysis of the uptime graph shows a drop on the 4$^{\text{th}}$ day followed by a degradation that reached a plateau on day 18 then was finally fixed on day 26. Looking at success-ratio instead of user-uptime paints the same picture, as can be seen in Fig. 18. Looking at either of these two graphs, it is reasonable to assume that this incident has a single cause.

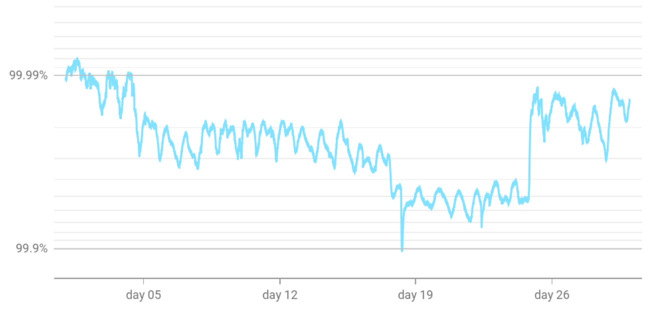


Figure 18: success-ratio of Google Drive, same 30 days as Fig. 17.

But if we plot success-ratio and user-uptime together as in Fig. 19, we see that they diverge on day 18. Indeed, an investigation showed that the period between day 18 and day 26 was caused by a different issue, introduced by an attempted fix for the first problem. It manifests differently in user-uptime compared to success-ratio due to different client behavior for this new issue. This divergence led the engineers to the correct path in fixing the new issue instead of trying to find out why the old issue was still ongoing, and made the path to resolution easier and quicker.

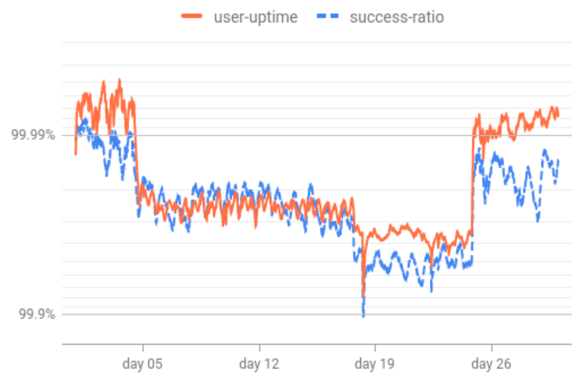


Figure 19: Divergence of user-uptime and success-ratio

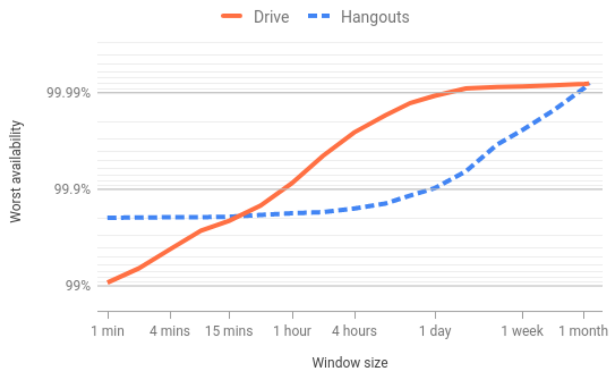


Figure 20: Monthly windowed user-uptime distinguishes between the nature of unavailability

6.5 Windowed uptime indicates burstiness of unavailability

When we look at availability metrics aggregated over a month or a quarter we cannot see short episodes of poor availability. Fig. 20 shows the windowed user-uptime of Drive and Hangouts. The two curves meet at the same point at the 1 month mark; thus they have the same scaled availability (99.972%) over the course of the month. Their windowed user-uptime graphs, however, tell different stories.

Windowed user-uptime reveals that it is a four hour episode (the knee of the curve) that held back Hangouts from having an even higher availability. In a different month without such an episode or if we fix the root cause behind such episodes, this product would have a higher availability.

For Drive, the windowed user-uptime curve has no pronounced knee; this means that rather than a single long episode, there are continuous short episodes which are holding back the service from having a higher availability. Thus, unless we fix their root cause, Drive will continue to suffer from these downtimes month after month.

By exposing shorter episodes of low availability (e.g., Hangouts' four hour episode), windowed user-uptime alerts us to problems that would otherwise be masked by the (commonly-used) monthly aggregation. Our teams consequently use windowed user-uptime to identify, root-cause and then fix the sources of these short episodes, improving overall availability.

7 Applicability of windowed user-uptime

To calculate windowed user-uptime we need fine-grained logs of individual user operation. These logs must include a key that enables us to chain together operations for each user, the timestamp of the operation and the status of each operation (success or failure). In some cases, additional information is invaluable: for example, (i) knowing the type of the operation enables us to determine if different operations have different availability and (ii) knowing the organization of a user enables us to determine if a particular organization is experiencing worse availability compared to other organizations.

In the simplest case we need to retain only the cumulative count of up and down minutes for each minute to calculate windowed user-uptime over any time duration. If we want to slice data along additional dimensions we must maintain the count of up and down minutes for each dimension (organization, operation type, etc.).

It took us about 1 year to deploy windowed user-uptime to all of the G Suite applications. This time included implementation (e.g., to normalize the different log formats so we can use the same pipeline across all of our applications and to build the pipeline for calculating the metric) but the bulk of the time was in determining which operations we should consider in windowed user-uptime and whether or not a given operation's availability could be visible to users. From this experience we are confident that windowed user-uptime is broadly applicable: any cloud service provider should be able to implement it.

8 Conclusion

We have introduced a novel availability metric, *user-uptime* which combines the advantages of per-user aggregation with those of using a time-based availability measure. We have shown that as a result user-uptime avoids multiple kinds of bias: hyper-active users contribute similarly to the metric as regular users, and even behavioral changes during an outage result in a proportional and meaningful measurement that in many cases is even more precise than success-ratio.

We have evaluated our metric against the commonly-used success-ratio metric using production data from G Suite services. We show that the bias in success-ratio is not an academic argument: we actually encounter it in production and that user-uptime avoids this bias.

We have introduced a visualization technique, windowed availability, that allows us to study multiple time-scales from single minutes to a full quarter in an easy to understand graph. Again using production data from G Suite services, we show that windowed user-uptime sheds invaluable and actionable insight into the availability of our services. Specifically, windowed user-uptime enables us to differentiate between many short and fewer but longer outages. This ability focuses our engineering efforts into improvements that will yield the most gains in user-perceived availability.

Acknowledgments

We are grateful for insighful contributions from André Prinsloo, Jesse Bickmore, Aaron Isotton, Andrés Martínez and Tony Scelfo, all of Google, and thoughtful comments from the anonymous reviewers.

References

[1] 5-minute outage costs google $545,000 in revenue. https://venturebeat.com/2013/08/16/3-minute-outage-costs-google-545000-in-revenue/.

[2] Amazon s3 service level agreement. https://aws.amazon.com/s3/sla/.

[3] Amazon.com goes down, loses $66,240 per minute. https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/.

[4] Cloud services you can trust: Office 365 availability. https://www.microsoft.com/en-us/microsoft-365/blog/2013/08/08/cloud-services-you-can-trust-office-365-availability/.

[5] G suite service level agreement. https://gsuite.google.com/intl/en/terms/sla.html.

[6] Google's bad week: Youtube loses millions as advertising row reaches us. https://www.theguardian.com/technology/2017/mar/25/google-youtube-advertising-extremist-content-att-verizon.

[7] Slack system status: How is uptime calculated? https://status.slack.com/.

[8] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, Renton, WA, 2018. USENIX Association.

[9] Salman Abdul Baset. Cloud slas: present and future. *Operating Systems Review*, 46, 2012.

[10] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.", 2016.

[11] Betsy Beyer, Niall Richard Murphy, David K. Rensin Rensin, Stephen Thorne, and Kent Kawahara. *The Site Reliability Workbook: Practical Ways to Implement SRE*. " O'Reilly Media, Inc.", 2018.

[12] John Brevik, Daniel Nurmi, and Rich Wolski. Quantifying machine availability in networked and desktop grid systems. Technical report, CS2003-37, University of California at Santa Barbara, 2003.

[13] Aaron B. Brown and David A. Patterson. Towards availability benchmarks: A case study of software RAID systems. In *Proceedings of the General Track: 2000 USENIX Annual Technical Conference*, 2000.

[14] David Chappell. Introducing the windows azure platform. *David Chappell & Associates White Paper*, 2010.

[15] Perry Cheng and Guy E Blelloch. A parallel, real-time garbage collector. *ACM SIGPLAN Notices*, 36(5):125–136, 2001.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[17] Matteo Dell'Amico, Maurizio Filippone, Pietro Michiardi, and Yves Roudier. On user availability prediction and network applications. *CoRR*, abs/1404.7688, 2014.

[18] Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. Back to the future: On predicting user uptime. *CoRR*, abs/1010.0626, 2010.

[19] Patricia Takako Endo, Moisés Rodrigues, Glauco Estacio Gonçalves, Judith Kelner, Djamel Fawzi Hadj Sadok, and Calin Curescu. High availability in clouds: systematic review and research challenges. *J. Cloud Computing*, 5, 2016.

[20] A. L. Goel and J. Soenjoto. Models for hardware-software system operational-performance evaluation. *IEEE Transactions on Reliability*, R-30(3), 1981.

[21] A. Goyal and S. S. Lavenberg. Modeling and analysis of computer system availability. *IBM Journal of Research and Development*, 31(6), 1987.

[22] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9), 1991.

[23] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4), 1997.

[24] ISO Iso. iec/ieee international standard-systems and software engineering–vocabulary. Technical report, ISO/IEC/IEEE 24765: 2017 (E), 2017.

[25] Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. Service level agreement in cloud computing. 2009.

[26] Lingshuang Shao, Junfeng Zhao, Tao Xie, Lu Zhang, Bing Xie, and Hong Mei. User-perceived service availability: A metric and an estimation approach. In *2009 IEEE International Conference on Web Services*, 2009.

[27] Ushio Sumita and Yasushi Masuda. Analysis of software availability/reliability under the influence of hardware failures. *IEEE Trans. Software Eng.*, 12(1), 1986.

[28] Maria Toeroe and Francis Tam. *Service availability: principles and practice*. John Wiley & Sons, 2012.

[29] Ben Treynor, Mike Dahlin, Vivek Rau, and Betsy Beyer. The calculus of service availability. *Communications of the ACM*, 60(9), 2017.

[30] S. R. Welke, B. W. Johnson, and J. H. Aylor. Reliability modeling of hardware/software systems. *IEEE Transactions on Reliability*, 44(3), Sep. 1995.

Appendix: Monotonicity

We prove Eq. (7) which was stated in Section 5 without proof.

$$\frac{k}{k+1}\text{MCR}(w) \leq \text{MCR}(w') \qquad kw < w'. \tag{7}$$

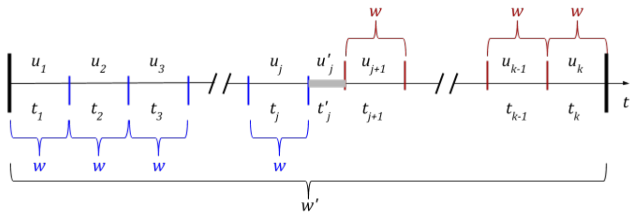


Figure 21: Embedding length-w windows in length-w' one. for $w' \neq kw$

To prove Eq. (7), we cover the larger window of size w' with windows of size w. Pick some $0 \leq j \leq k$ and fill the segment with j windows without a gap from the left and $k-j$ windows from the right, as in Fig. 21. As w' is not an integer multiple of w, this will leave a small segment uncovered in the middle. Like before, we write the availability of $\mathbf{w}'$ in terms of the uptime and total time of the segments as:

$$A(\mathbf{w}') = \frac{U}{T} = \frac{u_1 + \ldots + u_j + u'_j + u_{j+1} + \ldots + u_k}{t_1 + \ldots + t_j + t'_j + t_{j+1} + \ldots + t_k}.$$

The availability of each window of length w is bounded by MCR(w):

$$\frac{u_i}{t_i} \geq \text{MCR}(w).$$

Substituting $u_i \geq \text{MCR}(w)t_i$ and $u'_j \geq 0$ yields:

$$A(\mathbf{w}') \geq \text{MCR}(w)\frac{t_1 + \ldots + t_j + t_{j+1} + \ldots + t_k}{t_1 + \ldots + t_j + t'_j + t_{j+1} + \ldots + t_k}$$

or

$$TA(\mathbf{w}') \geq \text{MCR}(w)(T - t'_j).$$

Depending on how many windows we add on the left or on the right, j can be any integer between 0 (all windows on the right) and k (all windows on the left). Each of these possibilities yields the same inequality as above, they differ only in the respective value of t'_j. Let's add all of these $k+1$ inequalities:

$$(k+1)TA(\mathbf{w}') \geq \text{MCR}(w)(kT + T - \sum_{j=0}^{k} t'_j).$$

Note, however, that

$$T - \sum_{j=0}^{k} t'_j \geq 0$$

because t'_j are total times of non-overlapping intervals, all part of T. Substituting this yields the promised bound:

$$A(\mathbf{w}') \geq \frac{k}{k+1}\text{MCR}(w).$$

We see, therefore, that every window of length $w' > kw$ contains at least one window of size w with lower availability. Since there exists a window whose availability is MCR(w'), Eq. (7) follows.

Understanding, Detecting and Localizing Partial Failures in Large System Software

Chang Lou
Johns Hopkins University

Peng Huang
Johns Hopkins University

Scott Smith
Johns Hopkins University

Abstract

Partial failures occur frequently in cloud systems and can cause serious damage including inconsistency and data loss. Unfortunately, these failures are not well understood. Nor can they be effectively detected. In this paper, we first study 100 real-world partial failures from five mature systems to understand their characteristics. We find that these failures are caused by a variety of defects that require the unique conditions of the production environment to be triggered. Manually writing effective detectors to systematically detect such failures is both time-consuming and error-prone. We thus propose OmegaGen, a static analysis tool that automatically generates customized watchdogs for a given program by using a novel program reduction technique. We have successfully applied OmegaGen to six large distributed systems. In evaluating 22 real-world partial failure cases in these systems, the generated watchdogs can detect 20 cases with a median detection time of 4.2 seconds, and pinpoint the failure scope for 18 cases. The generated watchdogs also expose an unknown, confirmed partial failure bug in the latest version of ZooKeeper.

1 Introduction

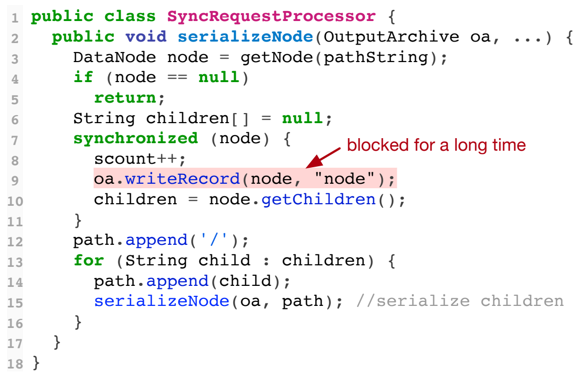
It is elusive to build large software that never fails. Designers of robust systems therefore must devise runtime mechanisms that proactively check whether a program is still functioning properly, and react if not. Many of these mechanisms are built with a simple assumption that when a program fails, it fails completely via crash, abort, or network disconnection.

This assumption, however, does not reflect the complex failure semantics exhibited in modern cloud infrastructure. A typical cloud software program consists of tens of modules, hundreds of dynamic threads, and tens of thousands of functions for handling different requests, running various background tasks, applying layers of optimizations, *etc.* Not surprisingly, such a program in practice can experience *partial failures*, where some, but not all, of its functionalities are broken. For example, for a data node process in a modern distributed file system, a partial failure could occur when a rebalancer thread within this process can no longer distribute unbalanced blocks to other remote data node processes, even though this process is still alive. Or, a block receiver daemon in this data node process silently exits, so the blocks are no longer persisted to disk. These partial failures are *not* a latent problem that operators can ignore; they can cause serious damage including inconsistency, "zombie" behavior and data loss. Indeed, partial failures are behind many catastrophic real-world outages [1, 17, 39, 51, 52, 55, 66, 85, 86]. For example, Microsoft Office 365 mail service suffered an 8-hour outage because an anti-virus engine module of the mail server was stuck in identifying some suspicious message [39].

When a partial failure occurs, it often takes a long time to detect the incident. In contrast, a process suffering a total failure can be quickly identified, restarted or repaired by existing mechanisms, thus limiting the failure impact. Worse still, partial failures cause mysterious symptoms that are incredibly difficult to debug [78], e.g., `create()` requests time out but `write()` requests still work. In a production ZooKeeper outage due to the leader failing partially [86], even after an alert was triggered, the leader logs contained few clues about what went wrong. It took the developer significant time to localize the fault within the problematic leader process (Figure 1). Before pinpointing the failure, a simple restart of the leader process was fruitless (the symptom quickly re-appeared).

Both practitioners and the research community have called attention to this gap. For example, the Cassandra developers adopted the more advanced accrual failure detector [73], but still conclude that its current design *"has very little ability to effectively do something non-trivial to deal with partial failures"* [13]. Prabhakaran *et al.* analyze partial failure specific to disks [88]. Huang *et al.* discuss the gray failure [76] challenge in cloud infrastructure. The overall characteristics of software partial failures, however, are not well understood.

In this paper, we first seek to answer the question, *how do partial failures manifest in modern systems?* To shed some light on this, we conducted a study (Section 2) of 100 real-world partial failure cases from five large-scale, open-source systems. We find that nearly half (48%) of the studied failures



```
 1 public class SyncRequestProcessor {
 2   public void serializeNode(OutputArchive oa, ...) {
 3     DataNode node = getNode(pathString);
 4     if (node == null)
 5       return;
 6     String children[] = null;
 7     synchronized (node) {
 8       scount++;
 9       oa.writeRecord(node, "node");
10       children = node.getChildren();
11     }
12     path.append('/');
13     for (String child : children) {
14       path.append(child);
15       serializeNode(oa, path); //serialize children
16     }
17   }
18 }
```

Figure 1: A production ZooKeeper outage due to partial failure [86].

cause certain software-specific functionality to be stuck. In addition, the majority (71%) of the studied failures are triggered by unique conditions in a production environment, *e.g.*, bad input, scheduling, resource contention, flaky disks, or a faulty remote process. Because these failures impact internal features such as compaction and persistence, they can be unobservable to external detectors or probes.

How to systematically detect and localize partial failures at runtime? Practitioners currently rely on running *ad-hoc* health checks (*e.g.*, send an HTTP request every few seconds and check its response status [3, 42]). But such health checks are too shallow to expose a wide class of failures. The state-of-the-art research work in this area is Panorama [75], which converts various requestors of a target process into observers to report gray failures of this process. This approach is limited by what requestors can observe externally. Also, these observers cannot localize a detected failure within the faulty process.

We propose a novel approach to construct effective partial failure detectors through *program reduction*. Given a program P, our basic idea is to derive from P a reduced but representative version W as a detector module and periodically test W in production to expose various potential failures in P. We call W an *intrinsic watchdog*. This approach offers two main benefits. First, as the watchdog is derived from and "imitates" the main program, it can more accurately reflect the main program's status compared to the existing stateless heartbeats, shallow health checks or external observers. Second, reduction makes the watchdog succinct and helps localize faults.

Manually applying the reduction approach on large software is both time-consuming and error-prone for developers. To ease this burden, we design a tool, *OmegaGen*, that statically analyzes the source code of a given program and generates customized intrinsic watchdogs for the target program.

Our insight for realizing program reduction in OmegaGen is that W's goal is *solely* to detect and localize runtime errors; therefore, it does not need to recreate the full details of P's business logic. For example, if P invokes `write()` in a tight loop, for checking purposes, a W with one `write()` may be sufficient to expose a fault. In addition, while it is tempting to check all kinds of faults, given the limited resources, W should focus on checking faults manifestable only in a production environment. Logical errors that deterministically lead to wrong results (*e.g.*, incorrect sorting) should be the focus of offline unit testing. Take Figure 1 as an example. In checking the `SyncRequestProcessor`, W need not check most of the instructions in function `serializeNode`, *e.g.*, lines 3–6 and 8. While there might be a slim chance these instructions would also fail in production, repeatedly checking them would yield diminishing returns for the limited resource budget.

Accurately distinguishing logically-deterministic faults and production-dependent faults in general is difficult. OmegaGen uses heuristics to analyze how "vulnerable" an instruction is based on whether the instruction performs some I/O, resource allocation, async wait, *etc.* So since line 9 of Figure 1 performs a write, it would be assessed as vulnerable and tested in W. It is unrealistic to expect W to always include the failure root cause instruction. Fortunately, a ballpark assessment often suffices. For instance, even if we only assess that the entire `serializeNode` function or its caller is vulnerable, and periodically test it in W, W can still detect this partial failure.

Once the vulnerable instructions are selected, OmegaGen will encapsulate them into checkers. OmegaGen's second contribution is providing several strong isolation mechanisms so the watchdog checkers do not interfere with the main program. For memory isolation, OmegaGen identifies the *context* for a checker and generates context managers with hooks in the main program which replicates contexts before using them in checkers. OmegaGen removes side-effects from I/O operations through redirection and designs an idempotent wrapper mechanism to safely test non-idempotent operations.

We have applied OmegaGen to six large (28K to 728K SLOC) systems. OmegaGen automatically generates tens to hundreds of watchdog checkers for these systems. To evaluate the effectiveness of the generated watchdogs, we reproduced 22 **real-world** partial failures. Our watchdogs can detect 20 cases with a median detection time of 4.2 seconds and localize the failure scope for 18 cases. In comparison, the best manually written baseline detector can only detect 11 cases and localize 8 cases. Through testing, our watchdogs exposed a new, confirmed partial failure bug in the latest ZooKeeper.

2 Understanding Partial Failures

Partial failures are a well known problem. Gupta and Shute report that partial failures occur much more commonly than total failures in the Google Ads infrastructure [70]. Researchers studied partial disk faults [88] and slow hardware faults [68]. But how software fails partially is not well understood. In this Section, we study real-world partial failures to gain insight into this problem and to guide our solution design.

Scope We focus on partial failure at the process granularity. This process could be standalone or one component in a large service (*e.g.*, a datanode in a storage service). Our studied partial failure is with respect to a process deviating from the functionalities it is supposed to provide *per se*, *e.g.*, store and

Software	Lang.	Cases	Ver.s (Range)	Date Range
ZooKeeper	Java	20	17 (3.2.1–3.5.3)	12/01/2009–08/28/2018
Cassandra	Java	20	19 (0.7.4–3.0.13)	04/22/2011–08/31/2017
HDFS	Java	20	14 (0.20.1–3.1.0)	10/29/2009–08/06/2018
Apache	C	20	16 (2.0.40–2.4.29)	08/02/2002–03/20/2018
Mesos	C++	20	11 (0.11.0–1.7.0)	04/08/2013–12/28/2018

Table 1: Studied software systems, the partial failure cases, and the unique versions, version and date ranges these cases cover.

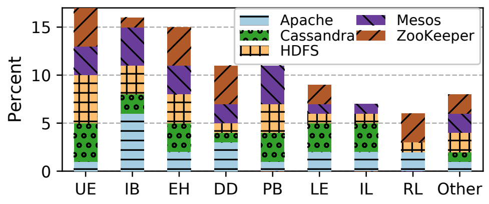


Figure 2: Root cause distribution. *UE*: uncaught error; *IB*: indefinite blocking; *EH*: buggy error handling; *DD*: deadlock; *PB*: performance bug; *LE*: logic error; *IL*: infinite loop; *RL*: resource leak.

balance data blocks, whether it is a service component or a standalone server. We note that users may define a partial failure at the service granularity (*e.g.*, Google drive becomes read-only), the underlying root cause of which could be either some component crashing or failing partially.

Methodology We study five large, widely-used software systems (Table 1). They provide different services and are written in different languages. To collect the study cases, we first crawl all bug tickets tagged with critical priorities in the official bug trackers. We then filter tickets from testing and randomly sample the remaining failures tickets. To minimize bias in the types of partial failures we study, we exhaustively examining each sampled case and manually determine whether it is a complete failure (*e.g.*, crash), and discard if so. In total, we collected 100 failure cases (20 cases for each system).

2.1 Findings

Finding 1: *In all the five systems, partial failures appear throughout release history (Table 1). 54%*[1] *of them occur in the most recent three years' software releases.*

Such a trend occurs in part because as software evolves, new features and performance optimizations are added, which complicates the failure semantics. For example, HDFS introduced a short-circuit local reads feature [30] in version 0.23. To implement this feature, a `DomainSocketWatcher` was added that watches a set of Unix domain sockets and invokes a callback when they become readable. But this new module can accidentally exit in production and cause applications performing short-circuit reads to hang [29].

Finding 2: *The root causes of studied failures are diverse. The top three (total 48%) root cause types are uncaught errors, indefinite blocking, and buggy error handling (Figure 2).*

Uncaught error means certain operation triggers some error condition that is not expected by the software. As an example, the streaming session in Cassandra could hang when the stream reader encounters errors other than `IOException` like `RuntimeException` [6]. Indefinite blocking occurs when some function call is blocked forever. In one case [27], the `EditLogTailer` in a standby HDFS namenode made an RPC `rollEdits()` to the active namenode; but this call was blocked when the active namenode was frozen but not crashed, which prevented the standby from becoming active. Buggy error handling includes silently swallowing errors, empty handlers [93], premature continuing, *etc.* Other common root causes include deadlock, performance bugs, infinite loop and logic errors.

[1]With sample size 100, the percents also represent the absolute numbers.

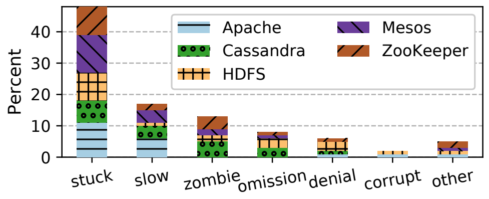


Figure 3: Consequence of studied failures.

Finding 3: *Nearly half (48%) of the partial failures cause some functionality to be stuck.*

Figure 3 shows the consequences of the studied failures. Note that these failures are all partial. For the *"stuck"* failures, some software module like the socket watcher was not making any progress; but the process was not completely unresponsive, *i.e.*, its heartbeat module can still respond *in time*. It may also handle other requests like non-local reads.

Besides "stuck" cases, 17% of the partial failures causes certain operation to take a long time to complete (the *"slow"* category in Figure 3). These slow failures are not just inefficiencies for optional optimization. Rather, they are severe performance bugs that cause the affected feature to be barely usable. In one case [5], after upgrading Cassandra 2.0.15 to 2.1.9, users found the read latency of the production cluster increased from 6 ms/op to more than 100 ms/op.

Finding 4: *In 13% of the studied cases, a module became a "zombie" with undefined failure semantics.*

This typically happens when the faulty module accidentally exits its normal control loop or it continues to execute even when it encounters some severe error that it cannot tolerate. For example, an unexpected exception caused the ZooKeeper listener module to accidentally exit its `while` loop so new nodes could no longer join the cluster [46]. In another case, the HDFS datanode continued even if the block pool failed to initialize [26], which would trigger a `NullPointerException` whenever it tried to do block reports.

Finding 5: *15% of the partial failures are silent (including data loss, corruption, inconsistency, and wrong results).*

They are usually hard to detect without detailed correctness specifications. For example, when the Mesos agent garbage collects old slave sandboxes, it could incorrectly wipe out the persistent volume data [37]. In another case [38], the Apache web server would "go haywire", *e.g.*, a request for a .js file would receive a response of image/png, because the backend connections are not properly closed in case of errors.

Finding 6: *71% of the failures are triggered by some specific environment condition, input, or faults in other processes.*

For example, a partial failure in ZooKeeper can only be triggered when some corrupt message occurs in the `length` field of a record [66]. Another partial failure in the ZooKeeper leader would only occur when a connecting follower hangs [50], which prevents other followers from joining the cluster. These partial failures are hard to be exposed by pre-production testing and require mechanisms to detect at runtime. Moreover, if a runtime detector uses a different setup or checking input, it may not detect such failures.

Finding 7: *The majority (68%) of the failures are "sticky".*

Sticky means the process will not recover from the faults by itself. The faulty process needs to be restarted or repaired to function again. In one case, a race condition caused an unexpected `RejectedExecutionException`, which caused the RPC server thread to silently exit its loop and stop listening for connections [9]. This thread must be restarted to fix the issue. For certain failures, some extra repair actions such as fixing a file system inconsistency [25] are needed.

The remaining (32%) failures are "transient", *i.e.*, the faulty modules could possibly recover after certain condition changes, *e.g.*, when the frozen namenode becomes responsive [27]. However, these non-sticky failures already incurred damage for a long time by then (15 minutes in one case [45]).

Finding 8: *The median diagnosis time is 6 days and 5 hours.*

For example, diagnosing a Cassandra failure [10] took the developers almost two days. The root cause turned out to be relatively simple: the `MeteredFlusher` module was blocked for several minutes and affected other tasks. One common reason for the long diagnosis time despite simple root causes is that the confusing symptoms of the failures mislead the diagnosis direction. Another common reason is the insufficient exposure of runtime information in the faulty process. Users have to enable debug logs, analyze heap, and/or instrument the code, to identify what was happening during the production failure.

2.2 Implications

Overall, our study reveals that partial failure is a common and severe problem in large software systems. Most of the studied failures are production-dependent (finding 6), which require runtime mechanisms to detect. Moreover, if a runtime detector can localize a failure besides mere detection, it will reduce the difficulty of offline diagnosis (finding 8). Existing detectors such as heartbeats, probes [69], or observers [75] are ineffective because they have little exposure to the affected functionalities internal in a process (e.g., compaction).

One might conclude that the onus is on the developers to add effective runtime checks in their code, such as a timer check for the `rollEdits()` operation in the aforementioned HDFS failure [27]. However, simply relying on developers to anticipate and add defensive checks for every operation is unrealistic. We need a *systematic* approach to help developers construct software-specific runtime checkers.

It would be desirable to completely automate the construction of customized runtime checkers, but this is extremely difficult in the general case given the diversity (finding 2) of partial failures. Indeed, 15% of the studied failures are silent, which require detailed correctness specifications to catch. Fortunately, the majority of failures in our study violate liveness (finding 3) or trigger explicit errors at certain program points, which suggests that detectors can be automatically constructed without deep semantic understanding.

3 Catching Partial Failures with Watchdogs

We consider a large server process π composed of many smaller modules, providing a set of functionalities R, *e.g.*, a datanode server with request listener, snapshot manager, cache manager, *etc.* A failure detector is needed to monitor the process for high availability. We target specifically partial failures. We define a partial failure in a process π to be when a fault does not crash π but causes safety or liveness violation or severe slowness for some functionality $R_f \subsetneq R$. Besides detecting a failure, we aim to localize the fault within the process to facilitate subsequent troubleshooting and mitigation.

Guided by our study, we propose an *intersection principle* for designing effective partial failure detectors—construct customized checks that intersect with the execution of a monitored process. The rationale is that partial failures typically involve specific software feature and bad state; to expose such failures, the detector need to exercise specific code regions with carefully-chosen payloads. The checks in existing detectors including heartbeat and HTTP tests are too generic and too *disjoint* with the monitored process' states and executions.

We advocate an **intrinsic watchdog** design (Figure 4) that follows the above principle. An intrinsic watchdog is a dedicated monitoring extension for a process. This extension regularly executes a set of checkers tailored to different modules. A watchdog driver manages the checker scheduling and execution, and optionally applies a recovery action. The key objective for detection is to let the watchdog experience similar faults as the main program. This is achieved through (a) executing *mimic-style* checkers (b) using *stateful* payloads (c) sharing execution environment of the monitored process.

Mimic Checkers. Current detectors use two types of checkers: *probe* checkers, which periodically invoke some APIs; *signal* checkers, which monitor some health indicator. Both are lightweight. But a probe checker can miss many failures because a large program has numerous APIs and partial failures may be unobservable at the API level. A signal checker is susceptible to environment noises and usually has poor accuracy. Neither can localize a detected failure.

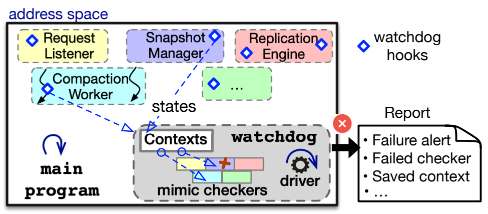


Figure 4: An intrinsic watchdog example.

We propose a more powerful *mimic-style* checker. Such checker selects some representative operations from each module of the main program, imitates them, and detects errors. This approach increases coverage of checking targets. And because the checker exercises code logic similar to the main program in production environment, it can accurately reflect the monitored process' status. In addition, a mimic checker can pinpoint the faulty module and failing instruction.

Synchronized States. Exercising checkers requires payloads. Existing detectors use *synthetic* input (e.g., fixed URLs [3]) or a tiny portion of the program state (e.g., heartbeat variables) as the payload. But triggering partial failures usually entails specific input and program state (§2). The watchdog should exercise its checkers with non-trivial state from the main program for higher chance of exposing partial failures.

We introduce *contexts* in watchdogs. A context is bound to each checker and holds *all* the arguments needed for the checker execution. Contexts are synchronized with the program state through hooks in the main program. When the main program execution reaches a hook point, the hook uses the current program state to update its context. The watchdog driver will not execute a checker unless its context is ready.

Concurrent Execution. It is natural to insert checkers directly in the main program. However, in-place checking poses an inherent tension—on the one hand, catching partial failures requires adding comprehensive checkers; on the other hand, partial failures only occur rarely, but more checkers would slow down the main program in normal scenarios. In-place checkers could also easily interfere with the main program through modifying the program states or execution flow.

We advocate watchdog to run *concurrently* with the main program. Concurrent execution allows checking to be decoupled so a watchdog can execute comprehensive checkers without delaying the main program during normal executions. Indeed, embedded systems domain has explored using concurrent watchdog co-processor for efficient error detection [84]. When a checker triggers some error, the watchdog also will not unexpectedly alter the main program execution. The concurrent watchdog should still live in the same address space to maximize mimic execution and expose similar issues, *e.g.*, all checkers timed out when the process hits long GC pause.

4 Generating Watchdogs with OmegaGen

It is tedious to manually write effective watchdogs for large programs, and it is challenging to get it right. Incautiously written watchdogs can miss checking important functions, alter the main execution, invoke dangerous operations, corrupt program states, *etc.* a watchdog must also be updated as the software evolves. To ease developers' burden, we design a tool, OmegaGen, which uses a novel *program reduction* approach to automatically generate watchdogs described in Section 3. The central challenge of OmegaGen is to ensure the generated watchdog accurately reflects the main program status without introducing significant overhead or side effects.

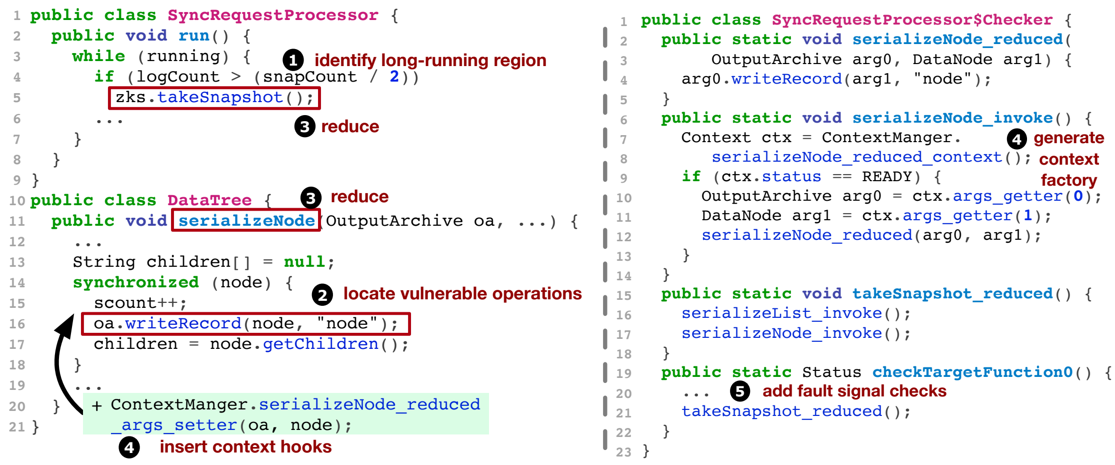
Overview and Target. OmegaGen takes the source code of a program *P* as an input. It finds the long-running code regions in *P* and then identifies instructions that may encounter production-dependent issues using heuristics and optional, user-provided annotations. OmegaGen encapsulates the vulnerable instructions into executable checkers and generates watchdog *W*. It also inserts watchdog hooks in *P* to update *W*'s contexts and packages a driver to execute *W* in *P*. Figure 5 shows an overview example of running OmegaGen.

As discussed in Section 2.2, it is difficult to automatically generate detectors that can catch all types of partial failures. Our approach targets partial failures that surface through explicit errors, blocking or slowness at certain instruction or function in a program. The watchdogs OmegaGen generates are particularly effective in catching partial failures in which some module becomes stuck, very slow or a "zombie" (*e.g.*, the HDFS `DomainSocketWatcher` thread accidentally exiting and affecting short-circuit reads). They are in general ineffective on *silent* correctness errors (*e.g.*, Apache web-server incorrectly re-using stale connections).

4.1 Identify Long-running Methods

OmegaGen starts its static analysis by identifying long-running code regions in a program (step ❶), because watchdogs only target checking code that is continuously executed. Many code regions in a server program are only for one-shot tasks such as database creation, and should be excluded from watchdogs. Some tasks are also either periodically executed such as snapshot or only activated under specific conditions. We need to ensure the activation of generated watchdog is aligned with the life span of its checking target in the main program. Otherwise, it could report wrong detection results.

OmegaGen traverses each node in the program call graph. For each node, it identifies potentially long-running loops in the function body, *e.g.*, `while(true)` or `while(flag)`. Loops with fixed iterations or that iterate over collections will be skipped. OmegaGen then locates all the invocation instructions in the identified loop body. The invocation targets are colored. Any methods invoked by a colored node are also recursively colored. Besides loops, we also support coloring periodic task methods scheduled through common libraries like `ExecutorService` in Java concurrent package. Note that this step may over-extract (*e.g.*, an invocation under a conditional). This is not an issue because the watchdog driver will check context validity at runtime (§4.4).



❶ identify long-running region
❸ reduce
❸ reduce
❷ locate vulnerable operations
❹ insert context hooks
❹ generate context factory
❺ add fault signal checks

```java
public class SyncRequestProcessor {
  public void run() {
    while (running) {
      if (logCount > (snapCount / 2))
        zks.takeSnapshot();
      ...
    }
  }
}
public class DataTree {
  public void serializeNode(OutputArchive oa, ...) {
    ...
    String children[] = null;
    synchronized (node) {
      scount++;
      oa.writeRecord(node, "node");
      children = node.getChildren();
    }
    ...
  }  + ContextManger.serializeNode_reduced
       _args_setter(oa, node);
}
```

```java
public class SyncRequestProcessor$Checker {
  public static void serializeNode_reduced(
      OutputArchive arg0, DataNode arg1) {
    arg0.writeRecord(arg1, "node");
  }
  public static void serializeNode_invoke() {
    Context ctx = ContextManger.
        serializeNode_reduced_context();
    if (ctx.status == READY) {
      OutputArchive arg0 = ctx.args_getter(0);
      DataNode arg1 = ctx.args_getter(1);
      serializeNode_reduced(arg0, arg1);
    }
  }
  public static void takeSnapshot_reduced() {
    serializeList_invoke();
    serializeNode_invoke();
  }
  public static Status checkTargetFunction0() {
    ...
    takeSnapshot_reduced();
  }
}
```

(a) A module in main program **(b)** Generated checker

Figure 5: Example of watchdog checker OmegaGen generated for a module in ZooKeeper.

A complication arises when a method has multiple call-sites, some of which are colored while others are not. Whether this method is long running or not depends on the specific execution. Moreover, an identified long-running loop may turn out to be short-lived in an actual run. To accurately capture the method life span and control the watchdog activation, OmegaGen designs a *predicate*-based algorithm. A *predicate* is a runtime property associated with a method which tracks whether a call site of this method is in fact reached.

For an invocation target inside a potentially long-running loop, a hook is inserted before the loop that sets its predicate and another hook after the loop that unsets its predicate. A callee of a potentially long-running method will have a predicate set to be equal to this caller's predicate. At runtime, the predicates are assigned and evaluated that activates or deactivates the associated watchdog. The predicate instrumentation occurs after OmegaGen finishes the vulnerable operation analysis (§4.2) and program reduction (§4.3).

4.2 Locate Vulnerable Operations

OmegaGen then analyzes the identified long-running methods and further narrows down the checking target candidates (step ❷). This is because even in those limited number of methods, a watchdog cannot afford to check all of their operations. Our study shows that the majority of partial failures are triggered by unique environment conditions or workloads. This implies that operations whose safety or liveness are heavily influenced by its execution environment deserve particular attention. In contrast, operations whose correctness is logically deterministic (*e.g.*, sorting), are better checked through offline testing or in-place assertions. Continuously monitoring such operations inside a watchdog would yield diminishing returns.

OmegaGen uses heuristics to determine for a given operation how *vulnerable* this operation is in its execution environment. Currently, the heuristics consider operations that perform synchronization, resource allocation, event polling, async waiting, invocation with external input argument, file or network I/O as highly vulnerable. OmegaGen identifies most of them through standard library calls. Functions containing complex while loop conditions are considered vulnerable due to potential infinite looping. Simple operations such as arithmetic, assignments, and data structure field accesses are tagged as not vulnerable. In the Figure 5a example, OmegaGen considers the `oa.writeRecord` to be highly vulnerable because its body invokes several write calls. These heuristics are informed by our study but can be customized through a rule table configuration in OmegaGen. For example, we can configure OmegaGen to consider functions with several exception signatures as vulnerable (i.e., potentially improperly handled). We also allow developers to annotate a method with a `@vulnerable` tag in the source code. OmegaGen will locate calls to the annotated method and treat them as vulnerable.

Neither our heuristics nor human judgment can guarantee that the vulnerable operation criteria are always sound and complete. If OmegaGen incorrectly assesses a safe operation as vulnerable, the main consequence is that the watchdog would waste resources monitoring something unnecessarily. Incorrectly assessing a vulnerable operation as risk-free is more concerning. But one nice characteristic of vulnerable operations is that they often propagate [67] – an instruction that blocks indefinitely would also cause its enclosing function to block; and, an instruction that triggers some uncaught error also propagates through the call stack. For example, in a real-world partial failure in ZooKeeper [66], even if OmegaGen misses the exact vulnerable instruction `readString`, a watchdog still has a chance to detect the partial failure if `dserialize` or even `pRequest` is assessed to be vulnerable. On the other hand, if a vulnerable operation is too high-level (*e.g.*, `main` is considered vulnerable), error signals can be swallowed internally and it would also make localizing faults hard.

4.3 Reduce Main Program

With the identified long-running methods and vulnerable operations, OmegaGen performs a top-down program reduction (step ❸) starting from the entry point of long-running methods. For example, in Figure 5a, OmegaGen will try to reduce the `takeSnapshot` function first. When walking the control flow graph of a method to be reduced, if an instruction is tagged as potentially vulnerable, it would be retained in the reduced method. Otherwise, it would be excluded. For a call instruction that is not tagged as vulnerable yet, it would be temporarily retained and OmegaGen will recursively try to reduce the target function. If eventually the body of a reduced method is empty, *i.e.,* no vulnerable operation exists, it will be discarded. Any call instructions that call this discarded method and were temporarily retained are also discarded.

The resulting reduced program not only contains all vulnerable operations reachable from long-running methods but also preserves the original structure, i.e., for a call chain `f` $\hookrightarrow$ `g` $\hookrightarrow$ `h` in the main program, the reduced call chain is `f'` $\hookrightarrow$ `g'` $\hookrightarrow$ `h'` This structure can help localize a reported issue. In addition, when later a watchdog invokes a validator (§4.6), the structure provides information on which validator to invoke.

If a type of vulnerable operation (*e.g.*, the `writeRecord` call in Figure 5a) is included multiple times in the reduced program, it could be redundant in terms of exposing failures. Therefore, OmegaGen will further reduce the vulnerable operations based on whether they have been included already. However, the same type of vulnerable operation may be invoked quite differently in different places, and only a particular invocation would trigger failure. If we are too aggressive in reducing based on occurrences, we may miss the fault-triggering invocation. So, by default OmegaGen only performs intra-procedural occurrence reduction: multiple `writeRecord` calls will not occur within a single reduced method but may occur across different reduced methods.

4.4 Encapsulate Reduced Program

OmegaGen will encapsulate the code snippets retained after step ❸ into watchdogs. But these code snippets may not be directly executable because of missing definitions or payloads. For example, the reduced version of `serializeNode` in Figure 5a contains an operation `oa.writeRecord(node, "node")`. But `oa` and `node` are undefined. OmegaGen analyzes *all* the arguments required for the execution of a reduced method. For each undefined variable, OmegaGen adds a local variable definition at the beginning of the reduced method. It further generates a *context factory* that provides APIs to manage all the arguments for the reduced method (step ❹). Before a variable's first usage in the reduced method, a getter call to the context factory is added to retrieve the latest value at runtime.

To synchronize with the main program, OmegaGen inserts hooks that call setter methods of the same context factory in the (non-reduced) method in the original program at the same point of access. The context hooks are further conditioned on the long-running predicate for this method (§4.1). When the watchdog driver executes a reduced method, it first checks whether the context is ready and skips the execution if the context is not ready. Together, context and predicate control the activation of watchdog checkers—only when the original program reaches the context hooks and the method is truly long-running would the corresponding operation be checked. For example, in the while loop of Figure 5a, if the log count has not reached the snapshot threshold yet, the predicate for `takeSnapshot` is true but the context for the reduced `serializeNode` is not ready so the checking is skipped.

4.5 Add Checks to Catch Faults

After step ❹, the encapsulated reduced methods can be executed in a watchdog. OmegaGen will then add checks for the watchdog driver to catch the failure signals from the execution of vulnerable operations in the reduced methods. OmegaGen targets both liveness and safety violations. Liveness checks are relatively straightforward to add. OmegaGen inserts a timer before running a checker. Setting good timeouts for distributed systems is a well-known hard problem. Prior work [82] argues that replacing end-to-end timeouts with fine-grained timeouts for local operations makes the setting less sensitive. We made similar observations and use a conservative timeout (default 4 seconds). Besides timeouts, the watchdog driver also records the moving average of checker execution latencies to detect potential slow faults.

To detect safety violations, OmegaGen relies on the vulnerable operations to emit explicit error signals (assertions, exceptions, and error codes) and installs handlers to capture them. OmegaGen also captures runtime errors, *e.g.*, null pointer exception, out of memory errors, `IllegalStateException`.

Correctness violations are harder to check automatically without understanding the semantics of the vulnerable operations. Fortunately such silent violations are not very common in our studied cases (§2). Nevertheless, OmegaGen provides a `wd_assert` API for developers to conveniently add semantic checks. When OmegaGen analyzes the program, it will treat `wd_assert` instructions as special vulnerable operations. It performs similar checker encapsulation (§4.4) by analyzing the context needed for such operations and generates checkers containing the `wd_assert` instructions. The original `wd_assert` in the main program will be rewritten as a no-op. In this way, developers can leverage the OmegaGen framework to perform concurrent expensive checks (*e.g.*, if the hashes of new blocks match their checksums) without blocking the main execution.

The watchdog driver records any detected error in a log file. The reported error contains the timestamp, failure type and symptom, failed checker, the corresponding main program location that the failed checker is testing. backtrace, *etc*. The watchdog driver also saves the context used by the failed checker to ease subsequent offline troubleshooting.

4.6 Validate Impact of Caught Faults

An error reported by a watchdog checker could be transient or tolerable. To reduce false alarms, the watchdog runs a validation task after detecting an error. The default validation is to simply re-execute the checker and compare, which is effective for transient errors. Validating tolerable errors requires testing software features. Note that the validator is *not* for handling errors but rather confirming impact. Writing such validation tasks mainly involves invoking some entry functions, *e.g.*, `processRequest(req)`, which is straightforward.

OmegaGen provides skeletons of validation tasks, and currently relies on manual effort to fill out the skeletons. But OmegaGen automates the decision of choosing which validation task to invoke based on which checker failed. Specifically, for a filled validation task T that invokes a function `f` in the main program, OmegaGen searches the generated reduced program structure (§4.3) in topological order and tries to find the first reduced method `m'` that either matches `f` or any method in the `f`'s callgraph. Then OmegaGen generates a hashmap that maps all the checkers that are rooted under `m'` to task T. At runtime, when an error is reported, the watchdog driver checks the map to decide which validator to invoke.

4.7 Prevent Side Effects

Context Replication. To prevent the watchdog checkers from accidentally modifying the main program's states, OmegaGen analyzes *all* the variables (context) referenced in a checker. It generates a replication setter in the checker's context manager, which will replicate the context when invoked. The replication ensures any modifications are contained in the watchdog's state. Using replicated contexts also avoids adding complex synchronization to lock objects during checking. But blindly replicating contexts will incur high overhead. We perform *immutability analysis* [74, 77] on the watchdog contexts. If a context is immutable, OmegaGen generates a reference setter instead, which only holds a reference to the context source.

To further reduce context replication, we use a simple but effective lazy copying approach that, instead of replicating a context upon each set, delays the replication to only when a getter needs it. To deal with potential inconsistency due to lazy replication—*e.g.*, the main program has modified the context after the setter call—we associate a context with several attributes: `version`, `weak_ref` (weak reference to the source object), and `hash` (hash code for the value of the source object). The lazy setter only sets these attributes but does not replicate the context. Later when the getter is invoked, the getter checks if the referent of `weak_ref` is not null. If so, it further checks if the current hash code of the referent's value matches the recorded `hash` and skip replication if they do not match (main program modified context). Besides the attribute checks in getters, the watchdog driver will check if the `version` attributes of each context in a vulnerable operation match and skip the checking if the versions are inconsistent (see further elaboration in Appendix A).

I/O Redirection and Idempotent Wrappers. Besides memory side effects; we also need to prevent I/O side effects. For instance, if a vulnerable operation is writing to a snapshot file, a watchdog could accidentally write to the same snapshot file and affect subsequent executions of the main program. OmegaGen adds I/O redirection capability in watchdogs to address this issue: when OmegaGen generates the context replication code, the replication procedure will check if the context refers to a file-related resource, and if so the context will be replicated with the file path changed to a watchdog test file under the same directory path. Thus watchdogs would experience similar issues such as degraded or faulty storage.

If the storage system being written to is internally load-balanced (*e.g.*, S3), however, the test file may get distributed to a different environment and thus miss issues that only affect the original file. This limitation can be addressed as our write redirection is implemented in a cloning library, so it is relatively easy to extend the logic of deciding the redirection path there to consider the load-balancing policy (if exposed). Besides, if the underlying storage system is layered and complex like S3, it is perhaps better to apply OmegaGen on that system to directly expose partial failures there.

For socket I/O, OmegaGen can perform similar redirection to a special watchdog port if we know beforehand the remote components are also OmegaGen-instrumented. Since this assumption may not hold, OmegaGen by default rewrites the watchdog's socket I/O operation as a ping operation.

If the vulnerable operation is a read-type operation, redirection to read from the watchdog special test file may not help. We design an idempotent wrapper mechanism so that both the main program and watchdog can invoke the wrapper safely. If the main program invokes the wrapper first, it directly performs the actual read-type operation and caches the result in a context. When the watchdog invokes the wrapper, if the main program is in the critical section, it will wait until the main program finishes, and then it gets the cached context. In the normal scenario, the watchdog can use the data from the read operation without performing the actual read. In the faulty scenario, if the main program blocks indefinitely in performing the read-type operation, the watchdog would uncover the hang issue through the timeout of waiting in its wrapper; a bad value from the read would also be captured by the watchdog after retrieving it. For each vulnerable operation of read-type, OmegaGen generates an idempotent wrapper with the above property, replaces the main program's original call instruction to invocation of the wrapper, and places a call instruction to the wrapper in the watchdog checker as well.

5 Implementation

We implemented OmegaGen in Java with 8,100 SLOC. Its core components are built on top of the Soot [90] program analysis framework, so it supports systems in Java bytecode. OmegaGen does not rely on specific JDK features. The Soot version we used can analyze bytecode up to Java 8. We leverage a cloning library [79] with around 400 SLOC of changes to support our selective context replication and I/O redirection mechanisms. OmegaGen's workflow consists of multiple phases to analyze and instrument the program and generate

	ZK	CS	HF	HB	MR	YN
SLOC	28K	102K	219K	728K	191K	229K
Methods	3,562	12,919	79,584	179,821	16,633	10,432

Table 2: Evaluated system software. *ZK*: ZooKeeper; *CS*: Cassandra; *HF*: HDFS; *HB*: HBase; *MR*: MapReduce; *YN*: Yarn.

	ZK	CS	HF	HB	MR	YN
Watchdogs	96	190	174	358	161	88
Methods	118	464	482	795	371	222
Operations	488	2,112	3,416	9,557	6,116	752

Table 3: Number of watchdogs and checkers generated. Not all watchdogs will be activated at runtime.

watchdogs. A single script automates the workflow and packages the watchdogs with the main program into a bundle.

6 Evaluation

We evaluate OmegaGen to answer several questions: (1) does our approach work for large software? (2) can the generated watchdogs detect and localize diverse forms of real-world partial failures? (3) do the watchdogs provide strong isolation? (4) do the watchdogs report false alarms? (5) what is the runtime overhead to the main program? The experiments were performed on a cluster of 10 cloud VMs. Each VM has 4 vCPUs at 2.3GHz, 16 GB memory, and 256 GB disk.

6.1 Generating Watchdogs

To evaluate whether our proposed technique can work for real-world software, we evaluated OmegaGen on six large systems (Table 2). We chose these systems because they are widely used and representative, with codebases as large as 728K SLOC to analyze. OmegaGen uses around 30 lines of default rules for the vulnerable operation heuristics (most are types of Java library methods) and an average of 10 system-specific rules (*e.g.*, special asynchronous wait patterns). OmegaGen successfully generates watchdogs for all six systems.

Table 3 shows the total watchdogs generated. Each watchdog here means a root of reduced methods. Note that these are static watchdogs. Only a subset of them will be activated in production by the watchdog predicates and context hooks (§4.1). We further evaluate how comprehensive the generated checkers are by measuring how many thread classes in the software have at least one watchdog checker generated. Figure 6 shows the results. OmegaGen achieves an average coverage ratio of 60%. For the threads that do not have checkers, they are either not long-running (*e.g.*, auxiliary tools) or OmegaGen did not find vulnerable operations in them. In general, OmegaGen may fail to generate good checkers for modules that primarily perform computations or data structure manipulations. The generated checkers may still contain some redundancy even after the reduction (§4.3).

6.2 Detecting Real-world Partial Failures

Failure Benchmark To evaluate the effectiveness of our generated watchdogs, we collected and reproduced 22 **real-world** partial failures in the six systems. Table 10 in the appendix lists the case links and types. All of these failures led to severe consequences. They involve sophisticated fault injection and workload to trigger. It took us 1 week on average to reproduce each failure. Seven cases are from our study in Section 2. Others are new cases we did not study before.

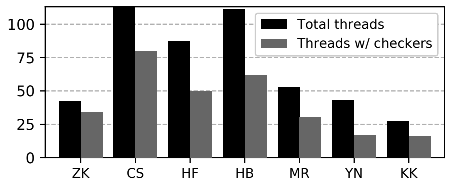


Figure 6: Thread-level coverage by generated watchdog checkers.

Detector	Description
Client (Panorama [75])	instrument and monitor client responses
Probe (Falcon [82])	daemon thread in the process that periodically invokes internal functions with synthetic requests
Signal	script that scans logs and checks JMX [40] metrics
Resource	daemon thread that monitors memory usage, disk and I/O health, and active thread count

Table 4: Four types of baseline detectors we implemented.

Baseline Detectors The built-in detectors (heartbeat) in the six systems cannot handle partial failures at all. We thus implement four types of advanced detectors for comparison (Table 4). The client checker is based on the observers in state-of-the-art work, Panorama [75]. The probe checker presents Falcon [82] app spies (which are also manually written in the Falcon paper). When implementing the signal and resource checkers, we follow the current best practices [15, 42] and monitor signals recommended by practitioners [2, 31, 41, 43].

Methodology The watchdogs and baseline detectors are all configured to run checks *every second*. When reproducing each case, we record when the software reaches the failure program point and when a detector first reports failure. The detection time is the latter minus the former. For slow failures, it is difficult to pick a precise start time. We set the start point using criteria recommended by practitioners, e.g., when number of outstanding requests exceeds 10 for ZooKeeper [31].

Result Table 5 shows the results. Overall, the watchdogs detected 20 out of the 22 cases with a median detection time of 4.2 seconds. 12 of the detected cases are captured by the default vulnerable operation rules. 8 are caught by system-specific rules. In general, the watchdogs were effective for liveness issues like deadlock, indefinite blocking as well as safety issues that trigger explicit error signals or exceptions. But they are less effective for silent correctness errors.

In comparison, as Table 5 shows, the best baseline detector only detected 11 cases. Even the combination of all baseline detectors detected only 14 cases. The client checkers missed 68% of the failures because these failures concern the internal functionality or some optimizations that are not immediately visible to clients. The signal checker is the most effective among the baseline detectors, but it is also noisy (§6.6).

Case Studies **ZK1** [45]: This is the running example in

	ZK1	ZK2	ZK3	ZK4	CS1	CS2	CS3	CS4	HF1	HF2	HF3	HF4	HB1	HB2	HB3	HB4	HB5	MR1	MR2	MR3	MR4	YN1
Watch.	4.28	-5.89	3.00	41.19	-3.73	4.63	46.56	38.72	1.10	6.20	3.17	2.11	5.41	7.89	✖	0.80	5.89	1.01	4.07	1.46	4.68	✖
Client	✖	2.47	2.27	✖	441	✖	✖	✖	✖	✖	✖	✖	✖	4.81	✖	6.62	✖	✖	✖	✖	8.54	7.38
Probe	✖	✖	✖	✖	15.84	✖	✖	✖	✖	✖	✖	✖	✖	4.71	✖	7.76	✖	✖	✖	✖	✖	✖
Signal	12.2	0.63	1.59	0.4	5.31	✖	✖	✖	✖	✖	✖	0.77	0.619	✖	0.62	61.0	✖	✖	✖	✖	0.60	1.16
Res.	5.33	0.56	0.72	17.17	209.5	✖	-19.65	✖	-3.13	✖	✖	0.83	✖	✖	✖	0.60	✖	✖	✖	✖	✖	✖

Table 5: Detection times (in seconds) for the real-world cases in Table 10. ✖: undetected.

	ZK1	ZK2	ZK3	ZK4	CS1	CS2	CS3	CS4	HF1	HF2	HF3	HF4	HB1	HB2	HB3	HB4	HB5	MR1	MR2	MR3	MR4	YN1
Watchdog	➻	➻	●	✻	➻	✻	●	✻	✻	✺	➻	➻	➻	➻	n/a	➻	✺	➻	➻	✺	➻	n/a
Client	n/a	●	●	n/a	●	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	●	n/a	❍	n/a	n/a	n/a	n/a	●	●
Probe	n/a	n/a	n/a	n/a	◗	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	◗	n/a	◗	n/a	n/a	n/a	n/a	n/a	n/a
Signal	●	➻	●	●	➻	n/a	n/a	n/a	n/a	n/a	n/a	➻	➻	n/a	✺	✺	n/a	n/a	n/a	n/a	➻	➻
Resource	●	●	●	●	●	n/a	●	n/a	●	n/a	n/a	●	n/a	n/a	n/a	●	n/a	n/a	n/a	n/a	n/a	n/a

Table 6: Failure localization for the real-world cases in Table 10.➻: pinpoint the faulty instr. ✻: pinpoint the faulty function or data structure. ✺: pinpoint a func in the faulty function's call chain. ◗: pinpoint some entry function in the program, which is distant from the root cause. ●: only pinpoint the faulty process. ❍: misleadingly pinpoint another innocent process. n/a: not applicable because failure is undetected.

the paper. A network issue caused a ZooKeeper remote snapshot dumping operation to be blocked in a critical section, which prevented update-type request processing threads from proceeding (Figure 1). OmegaGen generates a checker `serializeNode_reduced`, which exposed the issue in 4 s.

CS1 [7]: The Cassandra Commitlog executor accidentally died due to a bad commit disk volume. This caused the uncommitted writes to pile up, which in turn led to extensive garbage collection and the process entering a zombie status. The relevant watchdog OmegaGen generates is `CommitLogSegment_reduced`. Interestingly, this case had negative detection time. This happens because the executor successfully executed the faulty program point prior to the failure and set the watchdog context (log segment path). When the checker was scheduled, the context was still valid so the checker was activated and exposed the issue ahead of time.

HB5 [18]: Users observed some gigantic write-ahead-logs (WALs) on their HBase cluster even when WAL rolling is enabled. This is because when a peer is previously removed, one thread gets blocked for sending a shutdown request to a closed executor. Unfortunately this procedure holds the same lock `ReplicationSourceManager#recordLog`, which does the WAL rolling (to truncate logs). Our generated watchdog mimics the procedure of submitting request and waiting for completion, and experienced the same stalling issue on closed executor.

CS4 [11]: Due to a severe performance bug in the Cassandra compaction module, all the `RangeTombstones` ever created for the partition that have expired would remain in memory until the compaction completes. The compaction task would be very slow when the workloads contain a lot of overwrites to collections. The relevant checker OmegaGen generates is `SSTableWriter#append_reduced`. After the tombstones piles up, this checker reports a slow alert based on the dramatic ($10\times$) increase of moving average of operation latencies.

YN1 [44]: A new application (AM) was stuck after getting allocated to a recently added NodeManager (NM). This was caused by `/etc/hosts` on the ResourceManager (RM) not being updated, so this new NM was unresolvable when RM built the service tokens. RM would retry forever and the AM would keep getting allocated to the same NM. Our watchdogs failed to detect the issue. The reason is that the faulty operation `buildTokenService()` mainly creates some data structure, so OmegaGen failed to consider it as vulnerable.

6.3 Localizing Partial Failure

Detection is only the first step. We further evaluate the localization effectiveness for the detected cases in Table 5. we measure the distance between the error reporting location and the faulty program point. We categorize the distance into six levels of decreasing accuracy. Table 6 shows the result. Watchdogs directly pinpoint the faulty instruction for 55% (11/20) of the detected cases, which indicates the effectiveness of our vulnerable operation heuristics. In case MR1 [35], after noticing the symptom (reducer did not make progress for a long time), it took the user more than two days of careful log analysis and thread dumps to narrow down the cause. With the watchdog error report, the fault was obvious.

For 35% (7/20) of detected cases, the watchdogs either localize to some program point within the same function or some function along the call chain, which can still significantly ease troubleshooting. For example, in case HF2 [24], the balancer was stuck in a loop in `waitForMoveCompletion()` because `isPendingQEmpty()` will return false when no mover threads are available. The generated watchdog did not pinpoint either place. But it caught the error through timeout in executing a `future.get()` vulnerable operation in its checker `dispatchBlockMoves_reduced`, which narrows down the issue.

In comparison, the client or resource detectors can only pinpoint the faulty process. To narrow down the fault, users must spend significant time analyzing logs and code. In case HB4 [21], the client checker even blamed a wrong innocent process, which would completely mislead the diagnosis. The probe checker localizes failures to some internal functions in the program. But these functions are still too high-level and distant from the fault. The signal checker localizes 8 cases.

6.4 Fault-Injection Tests

To evaluate how the watchdogs may perform in real deployment, we conducted a random fault-injection experiment on

	ZK	CS	HF	HB	MR	YN
watch.	0–0.73	0–1.2	0	0–0.39	0	0–0.31
watch_v.	0–0.01	0	0	0–0.07	0	0
probe	0	0	0	0	0	0
resource	0–3.4	0–6.3	0.05–3.5	0–3.72	0.33–0.67	0–6.1
signal	3.2–9.6	0	0–0.05	0–0.67	0	0

Table 7: False alarm ratios (%) of all detectors in the evaluated six systems. Each cell reports the ratio range under three setups (stable, loaded, tolerable). *watch_v*: watchdog with validators.

the latest ZooKeeper. In particular, we inject four types of faults to the system: *Infinite loop* (modify loop condition to force running forever); *Arbitrary delay* (inject 30 seconds delay in some complex operations); *System resource contention* (exhaust CPU/memory resource); *I/O delay* (inject 30 seconds delay in file system or network). After that, we run a series of workloads and operations (e.g., restart some server). We successfully trigger 16 synthetic failures. Our generated watchdogs can detect 13 out of the 16 triggered synthetic failures with a median detection time of 6.1 seconds. The watchdogs pinpoint the injected failure scope for 11 cases.

6.5 Discovering A New Partial Failure Bug

During our continuous testing, our watchdogs exposed a new partial bug in the latest version (3.5.5) of ZooKeeper. We observe that our ZooKeeper cluster occasionally hangs and new create requests time out while the admin tool still shows the leader process is working. This symptom is similar to our studied bug ZK1. But that bug is already fixed in the latest version. The issue is also non-deterministic. Our watchdogs report the failure in 4.7 seconds. The watchdog log helps us pinpoint the root cause for this puzzling failure. The log shows the checker that reported the issue was `serializeAcls_reduced`. We further inspected this function and found that the problem was the server serializing the `ACLCache` inside a critical section. When developers fixed the ZK1 bug, this similar flaw was overlooked and recent refactoring of this class made the flaw more problematic. We reported this new bug [49], which has been confirmed by the developers and fixed.

6.6 Side Effects and False Alarms

We ran the watchdog-enhanced systems with extensive workloads and verified that the systems pass their own tests. We also verified the integrity of the files and client responses by comparing them with ones from the vanilla systems. If we disable our side-effect prevention mechanisms (§4.7), however, the systems would experience noticeable anomalies, *e.g.*, snapshots get corrupted, system crash; or, the main program would hang because the watchdog read the data from a stream.

We further evaluate the false alarms of watchdogs and baseline detectors under three setups: *stable*: runs fault-free for 12 hours with moderate workloads (§6.7); *loaded*: random node restarts, every 3 minutes into the moderate workloads, switch to aggressive workloads (3× number of clients and 5× request sizes); *tolerable*: run with injected transient errors tolerable by the system. Table 7 shows the results. The false alarm ratio is calculated from total false failure reports divided by the total number of check executions. Watchdogs did *not* report false alarms in the stable setup. But during a loaded period, they incur around 1% false alarms due to socket connection errors or resource contention. These false alarms would disappear once the transient faults are gone. With the validator mechanism (§4.6), the watchdog false alarm ratios (the *watch_v* row) are significantly reduced. Among the baseline detectors, we can see that even though signal checkers achieved better detection, they incur high false alarms (3–10%).

	ZK	CS	HF	HB	MR	YN
Analysis	21	166	75	92	55	50
Generation	43	103	130	953	131	89

Table 8: OmegaGen watchdog generation time (sec).

	ZK	CS	HF	HB	MR	YN
Base	428.0	3174.9	90.6	387.1	45.0	45.0
w/ Watch.	399.8	3014.7	85.1	366.4	42.1	42.3
w/ Probe.	417.6	3128.2	89.4	374.3	44.9	44.9
w/ Resource.	424.8	3145.4	89.9	385.6	44.9	44.6

Table 9: System throughput (op/s) w/ different detectors.

6.7 Performance and Overhead

We first measure the performance of OmegaGen's static analysis. Table 8 shows the results. For all but HBase, the whole process takes less than 5 minutes. HBase takes 17 minutes to generate watchdogs because of its large codebase.

We next measure the runtime overhead of enabling watchdogs and the baseline detectors. We used popular benchmarks configured as follows: for ZK, we used an open-source benchmark [16] with 15 clients sending 15,000 requests (40% read); for Cassandra, we used YCSB [61] with 40 clients sending 100,000 requests (50% read); for HDFS, we used built-in benchmark NNBenchWithoutMR which creates and writes 100 files, each file has 160 blocks and each block is 1MB; for HBase, we used YCSB with 40 clients sending 50,000 requests (50% read); for MapReduce and Yarn, we used built-in DFSIO benchmark which writes 400 10MB files.

Table 9 shows that the watchdogs incur 5.0%–6.6% overhead on throughput. The main overhead comes from the watchdog hooks rather than the concurrent checker execution. The probe detectors are more lightweight, incurring 0.2%–3.2% overhead. We also measure the latency impact. The watchdogs incur 9.3%–12.2% overhead on average latency and 8.3%–14.0% overhead on tail (99th percentile) latency. But given the watchdog's significant advantage in failure detection and localization, we believe its higher overhead is justified. For a cloud infrastructure, operators could also choose to activate watchdogs on a subset of the deployed nodes to reduce the overhead while still achieving good coverage.

We measure the CPU usages of each system w/o and w/ watchdogs. The results are 57%→66% (ZK), 199%→212% (CS), 33%→38% (HF), 36%→41% (HB), 5.6%→6.9% (MR), 1.5%→3% (YN). We also analyze the heap memory usages. The median memory usages (in MiB) are 128→131 (ZK), 447→459 (CS), 165→178 (HF), 197→201 (HB), 152→166

(MR), 154→157 (YN). The increase is small because contexts are only lazily replicated every checking interval, compared to continuous object allocations in the main program.

6.8 Sensitivity

We evaluate the sensitivity of our default 4-sec timeout threshold on detecting liveness issues with ZK1 [45] (stuck failure) and ZK4 [48] (slow failure). Under timeout threshold 100 ms, 300 ms, 500 ms, 1 s, 4 s, and 10 s, the detection times for ZK1 are respectively 0.51 s, 0.61 s, 0.70 s, 1.32 s, 4.28 s, and 12.09 s. The detection time generally decreases with smaller timeout, but it is bounded by the checking interval. With timeout of 100 ms, we observe 6 false positives in 5 minutes. For ZK4, when the timeout threshold is aggressive, the slow fault can be detected without the moving average mechanism (§4.5), in particular with detection times of 61.65 s (100 ms), 91.38 s (300 ms), 110.32 s (500 ms). Eventually the resource leak exhausts all available memory before the watchdog exceeds more conservative thresholds.

7 Limitations

OmegaGen has several limitations we plan to address in future work: (1) Our vulnerable operation analysis is heuristics-based. This step can be improved through offline profiling or dynamic adaptive selection. (2) Our generated watchdogs are effective for liveness issues and common safety violations. But they are ineffective to catch silent semantic failures. We plan to leverage existing resources that contain semantic hints such as test cases to derive runtime semantic checks. (3) OmegaGen achieves memory isolation with static analysis-assisted context replication. We will explore more efficient solutions like copy-on-write when porting OmegaGen to C/C++ systems. (4) OmegaGen generates watchdogs to report failures for individual process. One improvement is to pair OmegaGen with failure detector overlays [89] so the failure detector of one process could inspect another process' watchdogs. (5) Our watchdogs currently focus on fault detection and localization but not recovery. We will integrate microreboot [58] and ROC techniques [87].

8 Related Work

Partial failure has been discussed in multiple contexts. Arpaci-Dusseau and Arpaci-Dusseau propose the fail-stutter fault model [56]. Prabhakaran *et al.* analyze the fail-partial model for disks [88]. Correia *et al.* propose the ASC fault model [62]. Huang *et al.* propose a definition for gray failure in cloud [76]. Gunawi *et al.* [68] studies the fail-slow performance faults in hardware. Our study presented in Section 2 focuses on partial failures in modern cloud software. A recent work analyzes failures in cloud systems caused by network partitions [54]. Our work's scope is at the process granularity. A network partition may causes total failures to the partitioned processes (disconnected from other processes). Besides, our work covers much more diverse root causes beyond network issues.

Failure detection has been extensively studied [53, 59, 60, 63, 65, 71, 72, 80–82, 91]. But they primarily focus on detecting fail-stop failures in distributed systems; partial failures are beyond the scope of these detectors. Panorama [75] proposes to leverage observability in a system to detect gray failures [76]. While this approach can enhance failure detection, it assumes some external components happen to observe the subtle failure behavior. These logical observers also cannot isolate which part of the failing process is problematic, making subsequent failure diagnosis time-consuming [32].

Watchdog timers are essential hardware components found in embedded systems [57]. For general-purpose software, watchdogs are more challenging to construct manually due to the large size of the codebase and complex program logic. Consequently, existing software using the watchdog concept [4, 14] only designs watchdogs as shallow health checks (*e.g.*, http test) and a kill policy [42]. Our position paper [83] advocates for the intrinsic watchdog abstraction and articulates its design principles. OmegaGen provides the ability to automatically generate comprehensive, customized watchdogs for a given program through static analysis.

Several works aim to generate software invariants or ease runtime checking. Daikon [64] infers likely program invariants from dynamic execution traces. PCHECK [92] uses program slicing to extract configuration checks to detect latent misconfiguration during initialization. OmegaGen is complementary to these efforts. We focus on synthesizing checkers for monitoring long-running procedures of a program in production by using a novel program reduction technique.

9 Conclusion

System software continues to become ever more complex. This leads to a variety of partial failures that are not captured by existing solutions. This work first presents a study of 100 real-world partial failures in popular system software to shed light on the characteristics of such failures. We then present OmegaGen, which takes a program reduction approach to generate watchdogs for detecting and localizing partial failures. Evaluating OmegaGen on six large systems, it can generate tens to hundreds of customized watchdogs for each system. The generated watchdogs detect 20 out of 22 real-world partial failures with a median detection time of 4.2 seconds, and pinpoint the scope of failure for 18 cases; these results significantly outperform the baseline detectors. Our watchdogs also exposed a new partial failure in latest ZooKeeper.

Acknowledgments

We would like to thank the NSDI reviewers and our shepherd, Aurojit Panda, for their valuable comments that improved the paper. We thank Ziyan Wang for implementing the `wd_assert` API support for OmegaGen. We are grateful to the generous cloud research credits support from Azure, AWS and Google Cloud Platform. This work was supported in part by funding from NSF grants CNS-1755737 and CNS-1910133.

References

[1] Alibaba cloud reports IO hang error in north China. https://equalocean.com/technology/20190303-alibaba-cloud-reports-io-hang-error-in-north-china.

[2] Apache Cassandra: Some useful JMX metrics to monitor. https://medium.com/@foundev/apache-cassandra-some-useful-jmx-metrics-to-monitor-7f1d3ede294a.

[3] Apache module mod_proxy_hcheck. https://httpd.apache.org/docs/2.4/mod/mod_proxy_hcheck.html.

[4] Apache module mod_watchdog. https://httpd.apache.org/docs/2.4/mod/mod_watchdog.html.

[5] Cassandra-10477: java.lang.AssertionError in StorageProxy.submitHint. https://issues.apache.org/jira/browse/CASSANDRA-10477.

[6] Cassandra-5229: streaming tasks hang in netstats. https://issues.apache.org/jira/browse/CASSANDRA-5229.

[7] Cassandra-6364: Commit log executor dies and causes unflushed writes to quickly accumulate. https://issues.apache.org/jira/browse/CASSANDRA-6364.

[8] Cassandra-6415: Snapshot repair blocks forever if something happens to the remote response. https://issues.apache.org/jira/browse/CASSANDRA-6415.

[9] Cassandra-6788: Race condition silently kills thrift server. https://issues.apache.org/jira/browse/CASSANDRA-6788.

[10] Cassandra-8447: Nodes stuck in CMS GC cycle with very little traffic when compaction is enabled. https://issues.apache.org/jira/browse/CASSANDRA-8447.

[11] Cassandra-9486: LazilyCompactedRow accumulates all expired RangeTombstones. https://issues.apache.org/jira/browse/CASSANDRA-9486.

[12] Cassandra-9549: Memory leak in Ref.GlobalState due to pathological ConcurrentLinkedQueue.remove behaviour. https://issues.apache.org/jira/browse/CASSANDRA-9549.

[13] Cassandra: demystify failure detector, consider partial failure handling, latency optimizations. https://issues.apache.org/jira/browse/CASSANDRA-3927.

[14] Cloud computing patterns: Watchdog. http://www.cloudcomputingpatterns.org/watchdog/.

[15] Consul health check. https://www.consul.io/docs/agent/checks.html.

[16] Distributed database benchmark tester. https://github.com/etcd-io/dbtester.

[17] GoCardless service outage on October 10th, 2017. https://gocardless.com/blog/incident-review-api-and-dashboard-outage-on-10th-october.

[18] HBASE-16081: Removing peer in replication not gracefully finishing blocks WAL rolling. https://issues.apache.org/jira/browse/HBASE-16081.

[19] HBASE-16429: FSHLog deadlock if rollWriter called when ring buffer filled with appends. https://issues.apache.org/jira/browse/HBASE-16429.

[20] HBASE-18137: Empty WALs cause replication queue to get stuck. https://issues.apache.org/jira/browse/HBASE-18137.

[21] HBASE-21357: Reader thread encounters out of memory error. https://issues.apache.org/jira/browse/HBASE-21357.

[22] HBASE-21464: Splitting blocked with meta NSRE during split transaction. https://issues.apache.org/jira/browse/HBASE-21464.

[23] HDFS-11352: Potential deadlock in NN when failing over. https://issues.apache.org/jira/browse/HDFS-11352.

[24] HDFS-11377: Balancer hung due to no available mover threads. https://issues.apache.org/jira/browse/HDFS-11377.

[25] HDFS-12070: Failed block recovery leaves files open indefinitely and at risk for data loss. https://issues.apache.org/jira/browse/HDFS-12070.

[26] HDFS-2882: DN continues to start up, even if block pool fails to initialize. https://issues.apache.org/jira/browse/HDFS-2882.

[27] HDFS-4176: EditLogTailer should call rollEdits with a timeout. https://issues.apache.org/jira/browse/HDFS-4176.

[28] HDFS-4233: NN keeps serving even after no journals started while rolling edit. https://issues.apache.org/jira/browse/HDFS-4233.

[29] HDFS-8429: Error in DomainSocketWatcher causes others threads to be stuck threads. https://issues.apache.org/jira/browse/HDFS-8429.

[30] HDFS short-circuit local reads. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/ShortCircuitLocalReads.html.

[31] How to monitor Zookeeper. https://blog.serverdensity.com/how-to-monitor-zookeeper/.

[32] Just say no to more end-to-end tests. https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html.

[33] MapReduce-3634: Dispatchers in daemons get exceptions and silently stop processing. https://issues.apache.org/jira/browse/MAPREDUCE-3634.

[34] MapReduce-6190: Job stuck for hours because one of the mappers never started up fully. https://issues.apache.org/jira/browse/MAPREDUCE-6190.

[35] MapReduce-6351: Circular wait in handling errors causes reducer to hang in copy phase. https://issues.apache.org/jira/browse/MAPREDUCE-6351.

[36] MapReduce-6957: Shuffle hangs after a node manager connection timeout. https://issues.apache.org/jira/browse/MAPREDUCE-6957.

[37] Mesos-8830: Agent gc on old slave sandboxes could empty persistent volume data. https://issues.apache.org/jira/browse/MESOS-8830.

[38] mod_proxy_ajp: mixed up response after client connection abort. https://bz.apache.org/bugzilla/show_bug.cgi?id=53727.

[39] Office 365 update on recent customer issues. https://blogs.office.com/2012/11/13/update-on-recent-customer-issues/.

[40] Overview of the JMX technology. https://docs.oracle.com/javase/tutorial/jmx/overview/index.html.

[41] Running ZooKeeper in production. https://docs.confluent.io/current/zookeeper/deployment.html.

[42] Task health checking and generalized checks. http://mesos.apache.org/documentation/latest/health-checks.

[43] Tuning a database cluster with the performance service. https://docs.datastax.com/en/opscenter/6.1/opsc/online_help/services/tuneClusterPerfService.html.

[44] Yarn-4254: Accepting unresolvable NM into cluster causes RM to retry forever. https://issues.apache.org/jira/browse/YARN-4254.

[45] ZooKeeper-2201: Network issue causes cluster to hang due to blocking I/O in synch. https://issues.apache.org/jira/browse/ZOOKEEPER-2201.

[46] ZooKeeper-2319: UnresolvedAddressException cause the listener exit. https://issues.apache.org/jira/browse/ZOOKEEPER-2319.

[47] ZooKeeper-2325: Data inconsistency when all snapshots empty or missing. https://issues.apache.org/jira/browse/ZOOKEEPER-2325.

[48] ZooKeeper-3131: WatchManager resource leak. https://issues.apache.org/jira/browse/ZOOKEEPER-3131.

[49] ZooKeeper-3531: Synchronization on ACLCache cause cluster to hang. https://issues.apache.org/jira/browse/ZOOKEEPER-3531.

[50] ZooKeeper-914: QuorumCnxManager blocks forever. https://issues.apache.org/jira/browse/ZOOKEEPER-914.

[51] Twilio billing incident post-mortem: Breakdown, analysis and root cause. https://bit.ly/2V8rurP, July 23, 2013.

[52] Google compute engine incident 17008. https://status.cloud.google.com/incident/compute/17008, June 17, 2017.

[53] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS '09, pages 3–3, Monte Verità, Switzerland, 2009.

[54] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 51–68, Carlsbad, CA, USA, 2018.

[55] Amazon. AWS service outage on October 22nd, 2012. https://aws.amazon.com/message/680342.

[56] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HotOS '01, pages 33–. IEEE Computer Society, 2001.

[57] A. S. Berger. *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. CMP Books. Taylor & Francis, 2001.

[58] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI '04, pages 31–44, San Francisco, CA, 2004.

[59] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[60] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, May 2002.

[61] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, Indianapolis, Indiana, USA, 2010.

[62] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 41–41, Boston, MA, 2012.

[63] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 303–312. IEEE Computer Society, 2002.

[64] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, Los Angeles, California, USA, 1999.

[65] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2):99–112, Feb. 2003.

[66] E. Gilman. The discovery of Apache ZooKeeper's poison packet. https://www.pagerduty.com/blog/the-discovery-of-apache-zookeepers-poison-packet, May 7, 2015.

[67] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dussea, and B. Liblit. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, pages 14:1–14:16, San Jose, California, 2008.

[68] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 1–14, Oakland, CA, USA, 2018.

[69] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM SIGCOMM Conference*, SIGCOMM '15, pages 139–152, London, United Kingdom, 2015.

[70] A. Gupta and J. Shute. High-Availability at massive scale: Building Google's data infrastructure for Ads. In *Proceedings of the 9th International Workshop on Business Intelligence for the Real Time Enterprise*, BIRTE '15, 2015.

[71] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 175–188, Stevenson, Washington, USA, 2007.

[72] A. Haeberlen and P. Kuznetsov. The fault detection problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 99–114, Nîmes, France, 2009.

[73] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The φ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 66–78, Florianópolis, Brazil, 2004.

[74] B. Holland, G. R. Santhanam, and S. Kothari. Transferring state-of-the-art immutability analyses: Experimentation toolbox and accuracy benchmark. In *IEEE International Conference on Software Testing, Verification and Validation*, ICST '17, pages 484–491, March 2017.

[75] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018.

[76] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI. ACM, May 2017.

[77] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. Reim & ReImInfer: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 879–896, Tucson, Arizona, USA, 2012.

[78] D. King. Partial Failures are Worse Than Total Failures. https://www.tildedave.com/2014/03/01/application-failure-scenarios-with-cassandra.html, March 2014.

[79] K. Kougios. Java cloning library. https://github.com/kostaskougios/cloning.

[80] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 427–442, Lombard, IL, Apr. 2013.

[81] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 9:1–9:16, Bordeaux, France, 2015.

[82] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, Cascais, Portugal, Oct. 2011.

[83] C. Lou, P. Huang, and S. Smith. Comprehensive and efficient runtime checking in system software through watchdogs. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 51–57, Bertinoro, Italy, 2019.

[84] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors – a survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb 1988.

[85] Microsoft. Details of the December 28th, 2012 Windows Azure storage disruption in US south. `https://bit.ly/2Iofhcz`, January 16, 2013.

[86] D. Nadolny. Debugging distributed systems. In *SREcon 2016*, April 7-8 2016.

[87] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, and et al. Recovery oriented computing (ROC): Motivation, definition, techniques,. Technical report, USA, 2002.

[88] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 206–220, Brighton, United Kingdom, 2005.

[89] L. Suresh, D. Malkhi, P. Gopalan, I. P. Carreiro, and Z. Lokhandwala. Stable and consistent membership at scale with Rapid. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC '18, page 387–399, Boston, MA, USA, 2018.

[90] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[91] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 55–70, The Lake District, United Kingdom, 1998.

[92] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early detection of configuration errors to reduce failure damage. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 619–634, Savannah, GA, USA, 2016.

[93] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265, Broomfield, CO, 2014.

Appendix A Additional Clarifications

Consistency under Lazy Replication Section 4.7 describes that we associate a context with three attributes (`version`, `weak_ref`, and `hash`) to deal with potential inconsistency due to the lazy replication optimization. Here, we give a concrete example to clarify how potential inconsistency could arise and how it is addressed. With lazy replication (essentially "copy-on-get"), a context may be modified or even invalidated after the context setter call; if this occurs, the getter will replicate a different context value. For example,

```
       Main Program                     Watchdog Checker
-----------------------------     --------------------------
void foo() {                      void foo_reduced_invoke() {
  foo_reduced_args_setter(oa);
  write(oa);
  oa.append("test");
                  <---      oa = foo_reduced_ctx.args_getter(0);
}
```

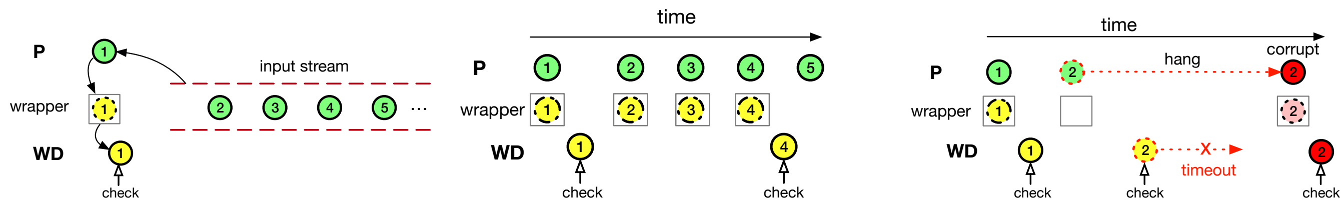
Id.	Root Cause	Conseq.	Sticky?	Study?
ZK1 [45]	Bad Synch.	Stuck	No	Yes
ZK2 [66]	Uncaught Error	Zombie	Yes	Yes
ZK3 [47]	Logic Error	Inconsist.	Yes	No
ZK4 [48]	Resource Leak	Slow	Yes	Yes
CS1 [7]	Uncaught Error	Zombie	Yes	Yes
CS2 [8]	Indefinite Blocking	Stuck	No	Yes
CS3 [12]	Resource Leak	Slow	Yes	No
CS4 [11]	Performance Bug	Slow	Yes	No
HF1 [29]	Uncaught Error	Stuck	Yes	Yes
HF2 [24]	Indefinite Blocking	Stuck	No	Yes
HF3 [23]	Deadlock	Stuck	Yes	No
HF4 [28]	Uncaught Error	Data Loss	Yes	No
HB1 [20]	Infinite Loop	Stuck	Yes	No
HB2 [19]	Deadlock	Stuck	Yes	No
HB3 [22]	Logic Error	Stuck	Yes	No
HB4 [21]	Uncaught Error	Denial	Yes	No
HB5 [18]	Indefinite Blocking	Silent	Yes	No
MR1 [35]	Deadlock	Stuck	Yes	No
MR2 [34]	Infinite Loop	Stuck	Yes	No
MR3 [36]	Improper Err Handling	Stuck	Yes	No
MR4 [33]	Uncaught Error	Zombie	Yes	No
YN1 [44]	Improper Err Handling	Stuck	Yes	No

Table 10: 22 real-world partial failures reproduced for evaluation. *ZK*: ZooKeeper; *CS*: Cassandra; *HF*: HDFS; *HB*: HBase; *MR*: MapReduce; *YN*: Yarn. Sticky?: whether the failure persists forever. Study?: whether the failure is from the studied cases in Section 2.

By the time the context getter is invoked in the checker, `oa` may already be invalidated (garbage collected). But since the getter will check the `weak_ref` attribute, it will find out the fact that the context is invalid (`weak_ref` returns `null`) and hence not replicate. If `oa` is still valid, the context getter will further check the hash code of the current value and skip replication if it does not match the recorded `hash`. This approach is lightweight. But it assumes the hash code contract of Java objects being honored in a program. If this is not the case, *e.g.*, `oa`'s hash code is the same regardless of its content, inconsistency (getter replicates a modified context) could arise. Such inconsistency may or may not cause an issue for the checker. For the above example, the checker's `write` may write `"xxxtest"` instead of `"xxx"` to the watchdog test file, which is still fine. But if another vulnerable operation has a special invariant on `"xxx"`, the inconsistency will lead to a false alarm at runtime. Our low false alarm rates during the 12-hour experiment period suggest that hash code contract violation is generally not a major concern for mature software.

Another consistency scenario to consider is when a checker uses some vulnerable operation that requires multiple context arguments. Since the context retrieval is asynchronous under the lazy replication optimization, a race condition could occur while a getter is retrieving all the arguments. For example,

```
          Main Program                          Watchdog Checker
-----------------------------------    --------------------------
```



(a) Idempotent wrapper for `readRecord` **(b)** Invoking wrapper in normal scenario **(c)** Invoking wrapper in faulty scenario

Figure 7: Illustration of idempotent wrapper

```
// called in a loop
synchronized void foo() {            void foo_reduced_invoke() {
                         <--- arg0 = foo_reduced_ctx.args_getter(0);
  ...
  foo_reduced_args_setter(oa, node);
  oa.writeRecord(node);
                         <---  arg1 = foo_reduced_ctx.args_getter(1);
}
```

After the getter retrieves `oa`, the second argument (`node`) is updated before the getter retrieves it. In this case, both arguments are valid and match their recorded `hash` attributes. However, they are mixed from two invocations of `foo()`. We address this inconsistency scenario with the `version` attributes. A checker will compare if the `version` attributes of all the contexts it needs are the same before invoking the checked operation, and skip the checking if the versions are inconsistent.

Appendix B Implementation Details

Idempotent Wrapper Section 4.7 describes our idempotent wrapper mechanism that allows watchdogs to safely invoke non-idempotent operations, especially `read`-type operations. We further elaborate the details for this mechanism here.

The basic idea is to have both the watchdog and main program invoke the wrapper instead of the original operation in a coordinated fashion. The wrapper distinguishes whether the call is from main program or the watchdog. Take a vulnerable operation `readRecord` as an example. In the fault-free scenario, the main program performs the actual `readRecord` like normal; the watchdog checker would get a cached value. In a faulty scenario, the main program may get stuck in `readRecord`; the watchdog would be blocked outside the critical section of the wrapper so it can detect the hang without performing the actual `readRecord`. Figure 7 illustrates both scenarios.

OmegaGen automatically generates idempotent wrappers for all read-type vulnerable operations. OmegaGen first locates all statements that invoke a read operation in the main program. It extracts the stream objects from these statements. A wrapper is generated for each type of stream object. The watchdog driver maintains a map between the stream objects and the wrapper instances. For the wrapper to later perform the actual operation, OmegaGen assigns a distinct operation number for each read-type method in the stream class, and generates a dispatcher that calls the method based on the op number. Then, OmegaGen replaces the original invocation with a call to the watchdog driver's wrapper entry point using the

		ZK	CS	HF	HB	MR	YN
Disk	Base	3.97	6.04	88.26	1.50	0.10	0.05
(MB/s)	w/ WD	4.04	6.12	89.02	1.53	0.10	0.05
Network	Base	997	2,884	27	993	1.3	1.5
(KB/s)	w/ WD	1,031	2,915	28	1,048	1.7	1.8

Table 11: Average disk and network I/O usages of the base systems and w/ watchdogs.

stream object, operation number, and caller source as the arguments. For example, `buf = istream.read();` in the main program would be replaced with `buf = WatchdogDriver.readHelp(istream, 1, 0);` where 1 is the op number for `read` and 0 means the wrapper is called from the main program.

The other steps in the checker construction for the read-type operations are similar to other types of vulnerable operations. The key difference is that OmegaGen will generate a self-contained checker for the *wrapped* operation instead of the operation. It particular, the checker OmegaGen generates will contain a call instruction to the proper wrapper using source 1 (from watchdog) as the argument.

Appendix C Supplementary Evaluation

Semantic Check API Our experiments in Section 6 did not use semantic checks, `wd_assert` (§4.5), to avoid biased results. But we did test using `wd_assert` on a hard case ZK3. Although the watchdog OmegaGen automatically generates detected this case, it is because the failure-triggering condition (bad disk) also affected some other vulnerable I/O operations in the watchdog. We wrote a `wd_assert` to check if the on-disk transaction records are far behind in-memory records:

```
wd_assert(lastProcessedZxid <= (new
ZKDatabase(txnLogFactory)).loadDataBase()+MISS_TXN_THRESHOLD);
```

OmegaGen handles the tedious details by automatically extracting the necessary context, encapsulating a watchdog checker, and removing this expensive statement from the main program. The resulted semantic checker can detect the failure within 2 seconds and pinpoint the issue.

I/O Usage Overhead We measured the disk I/O usages (using `iotop`) and network I/O usage (using `nethogs`) for the six systems with and without watchdogs under the same setup as our overhead experiment in Section 6.7. Table 11 shows the results. We can see the I/O usage increase incurred by the watchdogs is small (a median of 1.6% for disk I/O and 4.4% for network I/O).

Check before You Change: Preventing Correlated Failures in Service Updates

Ennan Zhai[†], Ang Chen[‡], Ruzica Piskac[○], Mahesh Balakrishnan[§,*]
Bingchuan Tian[♮], Bo Song[•], Haoliang Zhang[•]
[†]*Alibaba Group* [‡]*Rice University* [○]*Yale University* [§]*Facebook* [♮]*Nanjing University* [•]*Google*

Abstract

The reliability of cloud services can be significantly undermined by correlated failures due to shared service dependencies, even when the services are already replicated across machines. State-of-the-art failure prevention systems can proactively audit a service before its deployment to detect risks for correlated failures, but their auditing speeds are too slow for frequent service updates. This paper presents CloudCanary, a system that can perform *real-time* audits on service updates to identify the root causes of correlated failure risks, and generate improvement plans with increased reliability.

CloudCanary achieves this with two primitives, SNAPAUDIT and DEPBOOSTER. SNAPAUDIT leverages two insights to achieve high accuracy and efficiency: a) service updates typically affect only a small part of the service stack, allowing the majority of previous auditing results to be reused; and b) structural reliability auditing tasks can be reduced to a Boolean satisfiability problem, which can then be solved efficiently using modern SAT solvers. DEPBOOSTER, on the other hand, can generate improvement plans efficiently by reducing the required reasoning load, using novel techniques such as model counting. We demonstrate in our experiments that CloudCanary can perform audits over large deployments 200× faster than state-of-the-art systems, and that it consistently generates high-quality improvement plans within minutes. Moreover, CloudCanary can yield valuable insights over real-world traces collected from production environments.

1 Introduction

High reliability is an essential requirement for cloud services. To enhance reliability, cloud providers typically replicate states and functionality across multiple servers, under the assumption of failure independence [33, 34, 54].

Reality, however, is more complicated. The complex, multilayered nature of network/software stacks in cloud services may conceal underlying interdependencies between seemingly independent components, such as network switches and software modules. Failures of these common service dependencies can lead to correlated failures *despite replication*, causing service downtime [11, 25, 39, 70]. For example, a faulty top-of-rack (ToR) switch would affect all replicas in the same rack [18], and a buggy software component could propagate failures across all service instances it supports [38]. Such incidents have repeatedly made the headlines: in one of the Rackspace outage events [9], glitches in two core switches caused multiple servers to be inaccessible, leading to significant service disruption; in another incident, a single faulty data collector in Amazon EBS brought down the Relational Database service in an entire availability zone [3].

A number of previous efforts have focused on diagnosing the root causes of correlated failures [14, 23, 47, 58]. While this is useful, post-failure diagnostics typically involves prolonged failure recovery time [12, 64], as even the best of diagnostic tools cannot *prevent* service outages. Such outages can be quite costly: on average, a single datacenter outage can cause an economic loss of $740,357 [4].

More recent proposals aim to proactively prevent correlated failures by auditing the structural reliability of cloud services before deployment [22, 70, 71]. At a high level, these systems collect a comprehensive set of structural dependency data in cloud services, and construct a system-wide fault graph to encode the dependencies. They then identify potential risks for correlated failures from the fault graph.

However, state-of-the-art auditing systems are designed to perform audits at service initialization, not for conducting *real-time* audits throughout the service lifetime. Runtime audits are necessary, because existing work has shown that many dependencies potentially causing correlated failures are introduced by network and software updates (*e.g.*, reconfigurations and upgrades) during service runtime [39]. For example, a Gmail service upgrade configured microservice replicas to share the same vulnerable component, which later rendered user data unavailable for many hours [5]. Existing systems are impractical for real-time audits for two reasons.

- First, they are *too slow* in analyzing cloud-scale deployments in real time. For example, a state-of-the-art system takes ∼35 hours to analyze a 30,528-component service [70], making it only possible to perform a few audits per week. This cannot match the update frequency in today's clouds—for instance, Google reported 58 updates per week, roughly one update every three hours [37].
- Second, these tools can only alert the operator to correlated failure risks, but do not offer further support to find effective *improvement plans*. Thus, the operator needs to either manually reason about improvements to the existing deployment, or use automated tools to generate a plan from scratch [22, 70]. The former is error-prone, and the latter may result in a plan that requires considerable service reconfiguration. Moreover, both are inefficient.

[*] Work done while at Yale University.

In other words, although the operator may have enough lead time to perform audits at service initialization, the high turnaround time of existing systems prohibits their use in real-time auditing during service runtime.

We present CloudCanary, a system that can efficiently and accurately a) alert the operator to the root causes of correlated failure risks introduced by service updates, and b) generate a set of improved deployment plans with higher reliability. CloudCanary achieves this using two primitives—SnapAudit and DepBooster—to help prevent correlated failures during service runtime in a timely manner.

Contribution #1. SnapAudit (§3) can efficiently and accurately identify root causes for correlated failures in a given service snapshot. The design of SnapAudit addresses two challenges. The first is how to rapidly analyze a fault graph representing the service update snapshot. To address this challenge, we propose an *incremental auditing* algorithm to identify a set of *differential fault graphs*, which represents the "delta" between the service snapshots before and after an update. Based on the insight that service updates usually affect a *small* part of service stacks [37, 49, 60], extracting differential fault graphs enables us to avoid the need to re-analyze the entire fault graph from scratch. Second, although differential fault graphs are already smaller than the overall fault graph, analyzing each of them is still NP-hard and time-consuming [63]. We therefore propose an approach that speeds up the fault graph analysis by transforming a differential fault graph into a Boolean formula, and then solving the formula using a high-performance MinCostSAT solver [35].

Contribution #2. DepBooster (§4), on the other hand, helps the operator improve a risk-prone deployment. It allows the operator to specify a reliability goal (*e.g.*, the failure probability needs to be lower than a certain threshold), and then generates a set of alternative improvement plans that meet the specification. DepBooster also addresses two challenges. First, there are infinitely many potential improvement plans to be checked for their capability to satisfy the specified goal. To overcome this challenge, we utilize *network compression* [17]—a technique that can simplify a datacenter network by collapsing symmetric network structures and slicing away irrelevant parts—to significantly reduce the number of states we need to check. Second, even after compression, it still takes a long time to check whether a candidate deployment meets the specified goal. We further propose a novel algorithm based on model counting [20] for efficient checks.

To the best of our knowledge, CloudCanary is the first practical system capable of preventing correlated failure risks in service updates. We have built a CloudCanary prototype and evaluated it with a set of real-world scenarios (§6). Our results show that SnapAudit can identify correlated failure root causes in a 1,183,360-component service within 8 minutes, 200× faster than the state-of-the-art systems, and that DepBooster can find high-quality improvement plans within minutes.

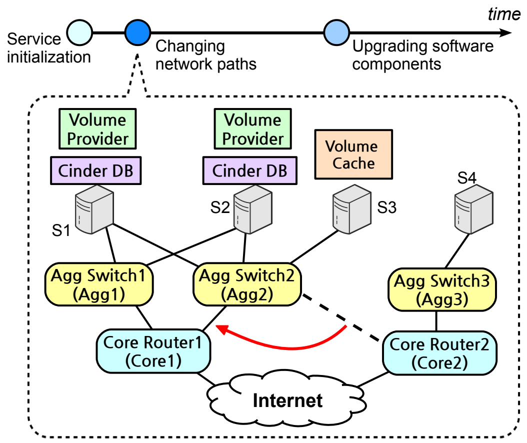


Figure 1: An update that affects the Cinder DB deployment, where the path Agg2→Core2 is shifted to Agg2→Core1 due to an ECMP configuration change.

2 Overview

In this section, we first motivate our problem further (§2.1). Then, we describe the state-of-the-art auditing systems and their limitations (§2.2 and §2.3). Finally, we present the architecture of CloudCanary (§2.4).

2.1 Motivation

Cloud operators ensure service reliability by replicating important state and functionality. Suppose that an operator deploys Cinder DB (a block storage system in OpenStack) in her datacenter, and that she replicates Cinder DB across multiple servers to increase reliability. Unbeknownst to this operator, the replicated Cinder DB instances may share deep dependencies, such as certain network or software components [3, 9]. The failures of such latent common dependencies can lead to a correlated failure across the entire system, undermining the use of replication. Such common dependencies are often called a *risk group*—a (small) number of components whose simultaneous failure results in a correlated failure.

To prevent correlated failures, the operator needs a tool to check for risk groups in a service deployment and generate improvement plans. For instance, if a risk group only contains one element, *e.g.*, a shared switch, it may potentially become a single point of failure. In this case, the operator may want to improve the deployment so that even the smallest risk group contains more than one element. In a similar spirit, if the estimated probability for correlated failures is above a threshold, the operator may want to find a functionally equivalent deployment with a lower failure probability.

Suppose that a service never goes through updates, then the above tasks only need to be performed once at service initialization. However, this is rarely the case in today's clouds, as most services experience frequent updates in their lifetime, and risk groups can be introduced in any of the updates [39]. Figure 1 shows an example: if the network path Agg2→Core2 is shifted to Agg2→Core1 (*e.g.*, due to a change to the ECMP configuration), such an update will introduce a new risk group $\sigma = \{\text{Core1}\}$, the fault of which will result in a correlated failure across both Cinder DB instances. Therefore, checking for risk groups and generating improvement plans need to be performed *continuously in real time*.

2.2 Starting Basis: Fault Graphs

Operators already apply a set of "golden standards" for increasing service reliability, such as rack-aware replica placement [8], geo-replication [1], canary tests [10], but achieving a comprehensive understanding of failure risks is a task that needs to be automated. To this end, several state-of-the-art auditing systems [22, 70, 71] have been proposed to proactively check for correlated failures. They do so using a common abstraction called a *fault graph* [63], which represents the structural dependencies of a service.

Fault graph. A fault graph is a layered DAG representing the logical relationships between component faults within a given system [63]. Figure 2 shows the fault graph of the example service in Figure 1. The fault graph has two types of nodes: *fault events* and *logic gates*. The leaf nodes in a fault graph are *basic faults*, which are the smallest units of failures under consideration, *e.g.*, the failure of a switch or software library. The root node in a fault graph represents a *target service fault*, which indicates the failure of the entire service. The rest of the nodes are *intermediate faults*, which describe how basic faults may cause larger service disruptions.

The fault propagation is encoded by layers of *logic gates* in between. If a component fails, the corresponding fault node outputs a 1 to its parent node, which could be either an `AND` or `OR` gate; otherwise the fault node outputs a 0. For an `OR` gate, if any of its children fail, a fault propagates upwards; for an `AND` gate, it only propagates a fault upwards if all of its children fail. Faulty nodes could be further associated with weights that encode the failure probabilities. Each non-leaf node has an *input gate* that connects its lower-layer faults, but leaf nodes, *i.e.*, basic faults, do not have an input gate.

Fault graph generation. State-of-the-art auditing systems (*e.g.*, INDaaS [70], reCloud [22]) have used existing data acquisition tools to automatically collect the structural dependency data needed for generating fault graphs. These tools cover a variety of dependency data, including network path dependencies [56, 70, 77], software component call flows [69, 73–75], and micro-service execution dependencies [24, 61]. Then, these auditing systems invoke various fault graph synthesis algorithms [51, 70–72] to automatically build fault graphs based on the acquired dependency data. Large-scale fault graph generation has been shown to be efficient—for example, INDaaS generates a 70,656-leaf fault graph within minutes [70].

Commercial data centers also deploy a variety of such profiling tools to track inter-service dependency, although the specific tools would differ from company to company. For instance, the Maelstrom [61] system at Facebook collects

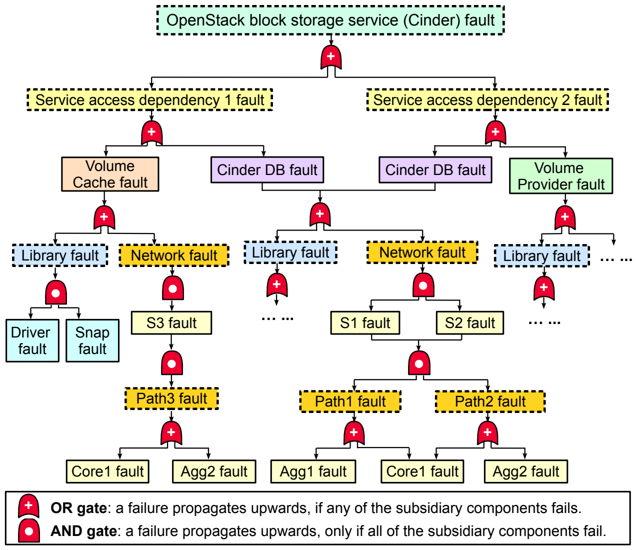


Figure 2: A fault graph representing the post-update service snapshot shown in Figure 1. Path 1, Path 2, and Path 3 represent the links Agg1→Core1, Agg2→Core1, and Agg2→Core1, respectively. Dashed boxes are logical components that do not exist physically.

service dependency data and uses it for failure mitigation. Later in our evaluation, we have also collected dependency data from a production data center using tools that are already in active deployment.

2.3 State of the Art and Limitations

State-of-the-art systems, such as INDaaS [70], reCloud [22], and RepAudit [71], can perform structural reliability audits on fault graphs to detect risk groups. They then output the identified risk groups to the operator to alert her to the risk. For instance, they may identify {Core1} to be a risk group, because its failure would cause the entire service to fail. However, existing systems all focus on *one-shot* audits at service initialization. They cannot handle *real-time* audits during service runtime due to the following two reasons.

Inefficient risk group auditing. Since detecting risk groups is NP-hard [63], existing auditing systems either perform an exhaustive search, which scales poorly to large deployments, or use heuristics, which sacrifices accuracy. For instance, INDaaS [70] takes ∼35 hours to analyze a 30,528-component service. Such speeds cannot match the frequency of network and software updates in today's clouds—for instance, Google reported 58 network updates per week [37].

Lack of support for generating improvement plans. Existing systems offer no support for the operator to automatically generate improvement plans. As a result, even after performing hours-long audits, the operator still needs to reason about improvement plans if the current service snapshot does not meet her reliability requirements. Existing systems such as INDaaS [70] and reCloud [22] can compute deployment plans from scratch, but such plans may differ considerably from

Table 1: Key techniques in CloudCanary.

Objective	Key techniques	Section(s)
Reusing previous audit results	Caching + Cache refreshes	3.1 + 3.4
Avoiding full-blown Cartesian products	Reduction to DNF (Disjunctive Normal Form) conversion	3.1
Incremental auditing	Differential fault graphs	3.2
Efficient auditing	Reduction to minimum-cost SAT	3.3
Avoiding large-scale Markov chains	Reduction to model counting	4.2
Handling non-uniform probabilities	Adding virtual leaf nodes	4.2
Reducing the search space	Network compression + Search heuristics	4.3

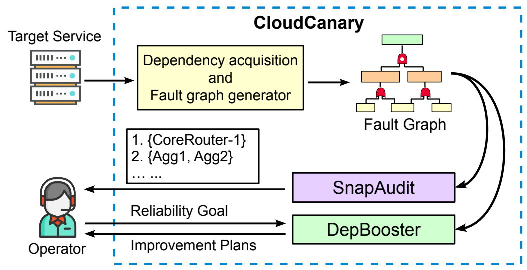


Figure 3: The workflow and architecture of CloudCanary with two novel primitives: SnapAudit and DepBooster.

the current snapshots and require non-trivial reconfiguration. Moreover, these systems are also inefficient to use in service runtime with high update frequency.

Therefore, although the two tasks can be performed with looser time constraints at service initialization, services with frequent updates demand better support for efficient audits and improvements in real time.

2.4 Our Approach: CloudCanary

We propose CloudCanary to achieve the above goals. Figure 3 shows CloudCanary's workflow. For a given service snapshot S, CloudCanary collects its dependency data and constructs a fault graph using existing dependency acquisition and fault graph generation modules [70]. The key innovation in CloudCanary is its two primitives SnapAudit and DepBooster. SnapAudit can extract risk groups from S, and DepBooster can generate improvement plans, both in a matter of minutes. Table 1 highlights the key techniques we have used and the objectives they are designed to achieve.

SnapAudit. To accelerate auditing, SnapAudit uses two insights. First, since service updates typically just affect a small subset of dependencies [37, 49, 60], there is no need to audit from scratch for each update. Rather, SnapAudit performs a complete fault graph analysis at service initialization, and aggressively reuses cached results to perform *incremental auditing* afterwards. Second, we use a novel encoding to reduce fault graph analysis to a *minimum cost Boolean Satisfiability (SAT) solving* problem, and leverage modern SAT solvers for fast auditing. This insight is driven by the fact that modern SAT solvers can solve complex Boolean formulas efficiently with accuracy guarantees.

DepBooster. The second primitive automatically generates improvement plans to meet a reliability goal, *e.g.*, the minimal risk group containing more than k elements, or the failure probability being lower than α. If naïvely done, assessing the failure probability requires solving a long Markov chain [63], and searching through all possible plans further exacerbates the inefficiency. We use a novel reduction to *model counting* to compute the failure probability, as well as a combination of *network compression* and *search heuristics* to reduce the search space.

3 The SnapAudit Design

This section details the design of SnapAudit that identifies the minimal risk groups in a given service snapshot. Figure 4 presents the key algorithms of SnapAudit: FirstAudit is only executed once at service initialization. IncAudit performs incremental auditing for the subsequent snapshots during service runtime. For a given service snapshot, the input of FirstAudit or IncAudit is a fault graph G representing its underlying dependency structure, and the output is Σ_G which contains the top-k *minimal* risk groups of G.

Minimal risk group. A risk group is *minimal* if the removal of any of its constituent elements makes it no longer a risk group. For instance, in Figure 2, there are two minimal risk groups: $\sigma_1 = \{\text{Core1}\}$ and $\sigma_2 = \{\text{Agg1} \wedge \text{Agg2}\}$. On the other hand, $\sigma_3 = \{\text{Agg1} \wedge \text{Core1}\}$ is also a risk group but is not a minimal risk group, because the failure of Core1 alone can cause the entire service to fail. Moreover, we can characterize a risk group's criticality by its cardinality, *e.g.*, σ_1 is more critical than σ_2 because $|\sigma_1| = 1 < |\sigma_2| = 2$—it takes two failures in σ_2 to take down the service but only a single failure in σ_1. The top-k risk groups of a given fault graph G are a ranked list of minimal risk groups by size or by failure probability. *e.g.*, $\Sigma_G = \{\sigma_1, \sigma_2\}$. Extracting minimal risk groups in a fault graph is NP-hard [63, 72].

3.1 The First Audit

At service initialization, we use FirstAudit to compute the risk groups from scratch. FirstAudit not only audits the overall fault graph G, but also *every subgraph* in G, thus enabling subsequent audits (performed by IncAudit) to reuse the results for these subgraphs. All audit results are recorded in a key-value cache Σ, where the key corresponds to a particular subgraph, and the value is its top-k minimal risk groups. FirstAudit builds a unique identifier for each subgraph by constructing a Merkle Hash Tree [53], and uses the root node's hash as the identifier of the entire subgraph. This allows for a

```
function FIRSTAUDIT($G$)
    if isleaf($G$) then
        $\Sigma_G \leftarrow \{G\}$
    for $c \in G.children$ do
        if $\Sigma_c = \emptyset$ then
            $\Sigma_c \leftarrow$ FIRSTAUDIT($c$)
    $\Sigma_G \leftarrow$ MERGE(G)
    return $\Sigma_G$
```

```
function MERGE($G$)
    $c_1, \cdots, c_k \leftarrow G.children$
    if $G.gate = AND$ then
        $\Sigma_G \leftarrow \text{DNF}(\Sigma_{c_1} \wedge \cdots \wedge \Sigma_{c_k})$
    else
        $\Sigma_G \leftarrow \text{DNF}(\Sigma_{c_1} \vee \cdots \vee \Sigma_{c_k})$
    return $\Sigma_G$
```

```
function INCAUDIT($G$)
    $\Pi \leftarrow$ GETBORDER($G$)
    for $t \in \Pi, t.children \notin \Pi$ do
        for $c \in t.children, \Sigma_c = \emptyset$ do
            $\Sigma_c \leftarrow$ MINCOSTSAT($c$)
    $\Sigma_G \leftarrow$ MERGEALL($G$)
    return $\Sigma_G$
```

```
function GETBORDER($G$)
    while $BFS(G)$ with $Q$ do
        $n \leftarrow Q.Pop()$
        if $\Sigma_n = \emptyset$ then
            if $c \in n.child, \Sigma_c \neq \emptyset$ then
                $L.append(n)$
            $Q.Push(n.children)$
    return $L$
```

Figure 4: The key functions in SNAPAUDIT: FIRSTAUDIT and MERGE (§3.1), INCAUDIT and GETBORDER (§3.2).

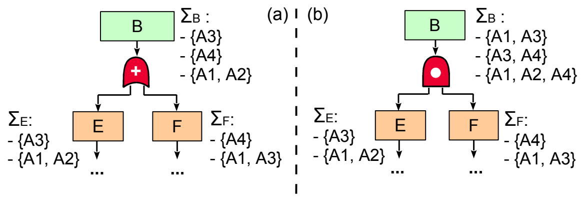


Figure 5: Merging risk groups for AND/OR gates.

more compact encoding of the subgraphs in the cache, given that the number of subgraphs in G is very large. For example, in Figure 5(a), the key of the subgraph rooted at B is $h(B) = h(h(E)||h(F))$, where h is a hash function and $||$ denotes concatenation. Indexing Σ by B's key would return $\Sigma_B = \{\{A3\}, \{A4\}, \{A1 \wedge A2\}\}$.

To generate Σ for both G and its subgraphs, a strawman solution is to directly call existing auditing systems such as INDaaS [70]. However, as discussed in §2.3, these systems are quite slow because their fault graph analysis algorithms scale poorly. Here, any inefficiency would be amplified several times over, because we are computing the minimal risk groups *for each subgraph* in G. To address this problem, we propose a completely different approach to computing the minimal risk groups, using high-performance Boolean formula translation toolchains such as Z3 [27] and Velev [62].

Overall, our FIRSTAUDIT algorithm starts with the leaf nodes, and recursively ascends to upper layers, until it reaches the root node of G. The base case for FIRSTAUDIT is to compute the minimal risk group list Σ_n for a leaf node n, where it simply returns $\Sigma_n = \{\{n\}\}$. In the inductive case, FIRSTAUDIT processes an intermediate node n with children $n_1, \cdots, n_k$ by calling MERGE on n and combining results for all its children. If n's children are connected by an OR gate, we have $\Sigma_n = \Sigma_{n_1} \cup \cdots \cup \Sigma_{n_k}$; otherwise, if n's children are connected by an AND gate we have $\Sigma_n = \Sigma_{n_1} \times \cdots \times \Sigma_{n_k}$, where $\times$ denotes Cartesian product. Figure 5 shows a concrete example.

Reduction to DNF conversion. A naïve MERGE over an AND gate requires a full-blown Cartesian product between risk groups, which leads to state explosion. If the size of each Σ_{n_i} is $|\Sigma|$, merging k of them would result in a set of size $|\Sigma|^k$; after s merges, the size would further grow to $|\Sigma|^{ks}$. To solve this problem, our insight is that MERGE can be achieved by a DNF (Disjunctive Normal Form) conversion, which can be efficiently computed using modern solvers [27]. A Boolean formula is in DNF if it is a disjunction of conjunctive clauses.

Consider the case shown in Figure 5(b), where we have $\Sigma_E = \{\{A3\}, \{A1 \wedge A2\}\}$ and $\Sigma_F = \{\{A4\}, \{A1 \wedge A3\}\}$. We need to compute $\Sigma_B = \Sigma_E \times \Sigma_F$, which can be transformed to a Boolean formula: $\phi = \Sigma_E \wedge \Sigma_F = ((A1 \wedge A2) \vee A3) \wedge ((A1 \wedge A3) \vee A4)$. By using Z3, we can quickly compute the DNF of ϕ, getting $(A1 \wedge A3) \vee (A1 \wedge A2 \wedge A4) \vee (A3 \wedge A4)$. As a result, Σ_B contains three minimal risk groups: $\{A1, A3\}$, $\{A3, A4\}$, and $\{A1, A2, A4\}$. Note that only DNF transformation can output all the minimal risk groups within one-run, and other solvers, *e.g.*, MinCostSAT, do not support such a capability.

3.2 Subsequent Audits

All subsequent audits are performed using INCAUDIT, which reuses the results in Σ generated by FIRSTAUDIT. As shown in Figure 4, INCAUDIT has three steps: GETBORDER, MINCOSTSAT, and MERGEALL. Given a fault graph G, INCAUDIT first uses GETBORDER to identify the differential fault graphs, and then invokes MINCOSTSAT to extract risk groups from each differential fault graph. Finally, INCAUDIT uses MERGEALL to merge the results for the differential fault graphs and those for the unchanged subgraphs, getting the final result Σ_G (*i.e.*, G's minimal risk groups).

GETBORDER. This step identifies a set of special *border nodes* that delineates the changed and unchanged portions of the fault graph. Concretely, a node n with children $n_1, \cdots, n_k$ is called a border node if a) at least one of $n_1 - n_k$'s key has a hit in Σ (*i.e.*, $\Sigma_{n_i} \neq \emptyset$), and b) at least one of them has a miss in Σ (*i.e.*, $\Sigma_{n_j} = \emptyset$). If all n's children have been previously audited, or none of them has been audited, then n is not a border node. For instance, in Figure 6, A and B are border nodes, but C–F are not.

To identify border nodes, we traverse G in a breadth-first order from the root. For each traversed node n, we check whether n has a hit in Σ. If n has a hit, we can reuse its result because 1) n is not a border node, and 2) n's subgraph has not changed. If n misses in Σ (*e.g.*, A in Figure 6), we check its children. If any of n's children has a hit in Σ (*e.g.*, D in Figure 6), we record n as a border node, and recurse and process n's children in order to find more border nodes.

We then extract *differential fault graphs* based on the border nodes and analyze such subgraphs from scratch, starting from the *bottom border node*. A node is a bottom border node if a) it is a border node, and b) none of its subgraphs contains

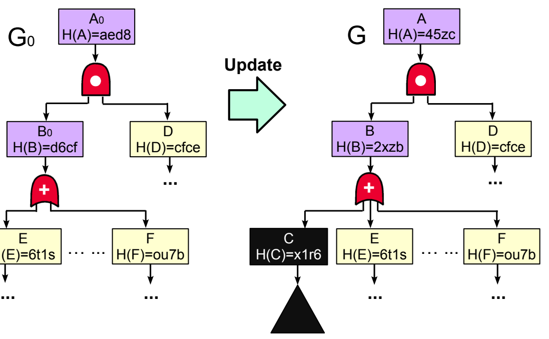


Figure 6: A service update where a subgraph C is added to the fault graph; this changes the keys for subgraphs rooted at A and B. In the updated fault graph, A and B are border nodes, B is the bottom border node, and D–F are unchanged. The subgraph rooted at C is a differential fault graph, which we invoke MINCOSTSAT on to obtain Σ_C. The results for Σ_D, Σ_E, and Σ_F have already been cached in Σ.

more border nodes. For example, in Figure 6, the only bottom border node is B; A is a border node, but not a bottom border node. We identify the bottom border nodes' children who miss in Σ (*e.g.*, C in Figure 6) as the roots of differential fault graphs, and analyze such subgraphs from scratch.

MINCOSTSAT. To analyze a differential fault graph, G_Δ, from scratch, a straightforward approach is to directly invoke FIRSTAUDIT. However, unlike the first audit at service initialization, which could be performed at leisure, INCAUDIT is frequently invoked during service runtime; thus, efficiency is much more critical. Thus, rather than audit all subgraphs in G_Δ, we only audit G_Δ itself. We achieve this by reducing this single audit to a *minimum-cost SAT* problem, which can be efficiently solved using modern SAT solvers. This step is denoted by MINCOSTSAT. For example, in Figure 6, because the subgraph rooted at C is a differential fault graph, we invoke MINCOSTSAT to compute its minimal risk groups, *i.e.*, Σ_C. We detail this MINCOSTSAT reduction in §3.3.

MERGEALL. After we use MINCOSTSAT to compute the risk groups for all differential fault graphs, we need to recompute the risk groups of G. Our insight is that we already have results for the siblings of these differential fault graphs (*e.g.*, D, E, and F in Figure 6), and we could directly use the DNF conversion in MERGE (§3.1), to obtain the risk groups for the entire G. Specifically, we generate a Boolean formula ϕ by only combining all the border nodes' children using their respective logic gates. Then, we transform ϕ into DNF, obtaining Σ_G. For example, in Figure 6, we first generate $\phi = (\Sigma_C \vee \Sigma_E \vee \Sigma_F) \wedge \Sigma_D$, and then transform it into DNF, getting the recomputed Σ_A.

3.3 The MinCostSat Solving

We now detail the design of MinCostSAT function, which can efficiently and accurately extract minimal risk groups from a given fault graph.

At a high level, a minimum-cost SAT problem [35] takes as input a Boolean formula ϕ with n Boolean variables $b_1, b_2, \ldots, b_n$, and a cost vector $\{w_i | w_i \geq 0, 1 \leq i \leq n\}$. The goal is to find a satisfying assignment to these variables such that ϕ evaluates to `True`, and simultaneously minimizing the following value: $W = \sum_{i=1}^{n} w_i b_i$.

We design the MINCOSTSAT function to compute the top-k risk groups. Initially, we transform an input fault graph into a Boolean formula ϕ, and initialize the cost of all the Boolean variables to one. For example in Figure 7, the fault graph at the left-hand can be transformed into (Agg1 $\vee$ Core1) $\wedge$ (Core1 $\vee$ Agg2). We then use a MinCostSAT solver to find the top-k critical risk groups through k rounds. Without loss of generality, for the i-th round, we identify the i-th smallest risk group in three steps: 1) we input the current formula ϕ_i and its cost vector into the MinCostSAT solver to generate the satisfying assignment with the minimal cost, 2) we obtain a risk group by extracting all the `True` literals from the resulting assignment, denoted as ψ. and 3) we use a conjunction to connect the current ϕ_i and the negation of ψ, generating a new $\phi_{i+1} = \phi_i \wedge \neg\psi$ for the next round.

3.4 Further Speedups

Since SNAPAUDIT heavily relies on the cache Σ for efficient audits, we propose two additional techniques to achieve further speedups. First, the results obtained during INCAUDIT can also be cached in Σ, so that Σ would grow over the service lifetime and the hit rate would improve. Second, INCAUDIT does not audit the subgraphs of a G_Δ, but the subgraphs may be needed for subsequent audits. We therefore run a background process that periodically invokes FIRSTAUDIT over the more recent snapshot to refresh the cache. However, we have omitted these techniques from the pseudocode for brevity.

4 The DEPBOOSTER Design

Identifying risk groups is a useful first step, but the operator still needs to reason about ways to increase the service reliability. Rather than ask the operator to achieve this manually, CloudCanary offers a second primitive, DEPBOOSTER, to generate improvement plans in an automated fashion.

4.1 The DEPBOOSTER Workflow

DEPBOOSTER offers the operator an interface to specify "reliability goals", and assesses if the current deployment meets the goals. If not, DEPBOOSTER generates improvement plans with increased reliability. These goals are specified as $spec = req \wedge action \wedge cons$. req is a requirement parameter. It can be a) $rcg > t$, which means the smallest risk groups in the deployment should contain more than t elements, b) $fp < \alpha$, which means the failure probability should be lower than some threshold α, or c) a combination of both. While DEPBOOSTER currently only supports constraints like failure probability and the size of risk groups, more constraints, *e.g.*, key paths, are easily added.

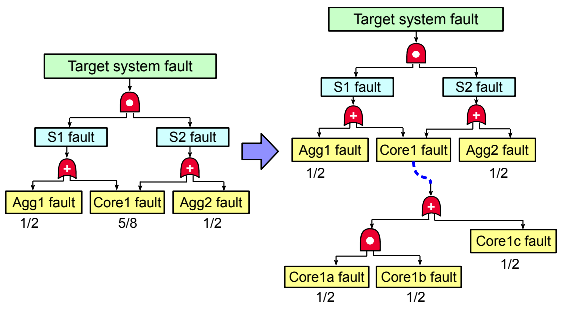


Figure 7: Transforming G by adding virtual leaf nodes.

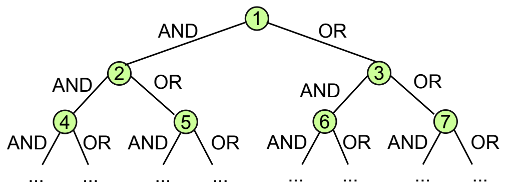


Figure 8: DEPBOOSTER searches through combinations of AND/OR gates to approximate a given probability.

DEPBOOSTER first assesses whether $rcg > t$ and $fp < \alpha$ already hold on the current snapshot. (Computing whether $rcg > t$ is achieved using SNAPAUDIT, which we described in §3; we defer the algorithm for computing whether $fp < \alpha$ to §4.2.) If both predicates hold, DEPBOOSTER reports so and terminates. Otherwise, it uses the strategies in *action* and the constraints in *cons* to generate improvement plans. *action* specifies an extensible set of basic actions to generate improvement plans with. Currently, DEPBOOSTER supports 1) `mov{r, A`$\rightarrow$`B}`, which moves a service replica `r` from a node `A` to another node `B`; 2) `add{r, A}`, which instantiates an additional replica `r` on node `A`; and 3) `link{A, B}`, which adds a network link between network components A and B. DEPBOOSTER then performs a search for improvement plans based on the basic actions. *cons* contains positive and negative constraints which specify that certain components must or must not be used in an improvement plan.

Example. We now provide a concrete example based on the scenario in Figure 1. Here, the operator provides DEPBOOSTER with a goal: $spec = \{rcg > 1 \wedge fp < 0.08\} \wedge \{mov\} \wedge \{Agg3\}$, which specifies that a) the smallest risk groups should contain more than one elements, and b) the failure probability should be lower than 0.08. If the current snapshot does not meet either of these two goals, DEPBOOSTER will generate a set of improvement plans. Moreover, the *spec* requires DEPBOOSTER to generate improvement plans by only moving replicated instances from the current replica servers to other servers. Finally, any improvement plan must still use the switch Agg3, as specified in *cons*. For this specification, DEPBOOSTER has generated two potential plans: a) `mov{CinderDB, S1->S4}`, and b) `mov{CinderDB, S2->S4}`. In other words, if we migrate the Cinder DB instance on S1 or S2 to S4, then new deployment would meet the desired goals.

4.2 Computing Failure Probability

We now describe how DEPBOOSTER computes the failure probability of a service snapshot. A strawman solution would be to derive the failure probability of the root node from these of the leaf nodes step by step, which is equivalent to computing the conditional probability of a Markov chain [63]. As we will show later, this is a time-consuming operation over large deployments, infeasible to be performed in real time. Instead, DEPBOOSTER uses two techniques to address this.

Technique #1: Model counters. DEPBOOSTER sidesteps the need for Markov chain computation by encoding this into a *model counting* problem. Suppose that the Boolean formula of the fault graph G is ϕ. A model counter [20] can find M—the number of satisfying assignments of ϕ. Assuming for now that all leaf nodes have a failure probability of exactly $\frac{1}{2}$, then the failure probability of G is simply $M/2^n$, where n is the number of leaf nodes in G. Since model counting does *not* need to compute the solutions themselves, but only the number of satisfying assignments, this is much more efficient than solving a Markov chain.

Technique #2: Virtual leaf nodes. However, another challenge arises: in practice, not all leaf nodes have the same failure probability, and such a probability is typically much lower than $\frac{1}{2}$. We address this by adding "virtual nodes" in the fault graph and reducing the problem again into the plain version of model counting. At a high level, we achieve this by substituting a node with failure probability of p with a subtree of virtual nodes, where all virtual nodes have a failure probability of $\frac{1}{2}$, and the failure probability of the entire virtual subtree evaluates to p with a user-defined precision ε. For instance, in the example shown in Figure 7, our goal would be to transform the node with $p = \frac{5}{8}$ (*i.e.*, `Core1 fault`) into a virtual subtree.

Figure 8 shows the solution space that DEPBOOSTER searches through to find a combination of gates that approximates a given failure probability. At any point in the search, the path from the root to a node n represents the current combination of gates. These gates further connect virtual nodes of failure probability $\frac{1}{2}$ (not shown in the figure). For instance, the path from the root to node 5 consists of an `AND` gate and then an `OR` gate, so the formula would be `(v1 AND v2) OR v3`, where `v1-v3` are virtual nodes with a failure probability of $\frac{1}{2}$. The failure probability of n_5 can then be computed as $p(n_5) = ((\frac{1}{2} \times \frac{1}{2}) + \frac{1}{2}) - (\frac{1}{2} \times \frac{1}{2}) \times \frac{1}{2} = \frac{5}{8}$. Our algorithm performs a BFS over the solution space, and constructs a combination of gates based on the path from the root to the current node. If $|p(n_j) - p| < \varepsilon$ holds for the current node, the search stops and we use the current combination to approximate a given probability. This transformation converts the fault graph G to a larger fault graph G' with (roughly) the same

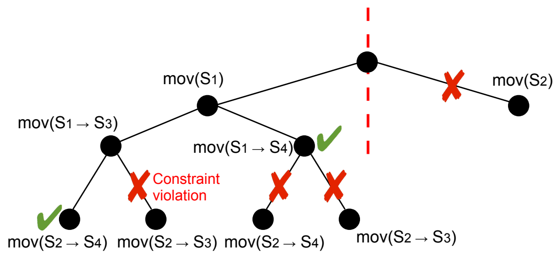


Figure 9: An example state-space tree.

failure probability that can be solved by model counters—*i.e.*, $p = M'/2^{n'}$, where M' is the model counter output for G' and n' is the number of leaf nodes in G'.

4.3 The DepBooster Algorithm

If the current deployment already meets the reliability goals, DepBooster directly terminates. Otherwise, it generates improvement plans by searching through a state-space tree. Each node in this tree represents one concrete move in *action*, and a path from the root to a leaf represents an improvement plan. Since there could be a large number of possible improvement plans, DepBooster uses two techniques to accelerate the search. Below, we use Figure 9 as an example to illustrate how DepBooster generates improvement plans for our running example in §4.1.

Technique #3: Network compression. We use the observation that datacenter network topologies tend to be highly symmetric, and can be "simplified" to equivalent topologies much smaller in size [17]. This enables DepBooster to perform the search on the smaller networks, and then map the solution back onto the original topologies. Driven by this observation, DepBooster transforms the input network topology D to a simplified topology d while preserving its original connectivity and reachability. Briefly, this is achieved by collapsing symmetric network structures (*i.e.*, routers and paths) and slicing away irrelevant structures. For instance, Figure 9 shows how the symmetric branch at S_2 has been pruned. We refer interested readers to the original paper [17] for proofs.

Technique #4: Iterative deepening. DepBooster then generates the state-space tree T_d based on the simplified topology d. It never materializes T_d in its entirety, but only explores it step by step. Concretely, DepBooster performs an Iterative Deepening Depth-First Search (IDDFS) [46] on T_d starting from the root. For each traversed node n, DepBooster checks whether n or any of n's children violates the specified constraints. If any constraint is violated, then the corresponding branches are pruned. For example, in Figure 9, the branch `mov(S2 -> S3)` under `mov(S1 -> S3)` is pruned because moving Cinder DB instances on S1 and S2 to S3 violates the constraint that Agg3 must be used in the new deployment. For the remaining nodes, DepBooster runs IncAudit and the failure probability computation approach (designed in §4.2) to check whether the size of risk groups and failure probability meet the specified goals. For instance, in Figure 9, we do not need to check any branches below the state `mov(S1 -> S4)`, since moving the Cinder DB instance on S1 to S4 has already satisfied the specified goals.

DepBooster can be configured to a) produce improvement plans with the smallest number of actions, b) find the first t improvement plans, and c) run until a timeout occurs. In the running example, we have used b) to find four improvement plans: 1) `mov{CinderDB, S1->S4}`, 2) `mov{CinderDB, S2->S4}`, 3) `mov{CinderDB, S1->S3}`, `mov{CinderDB, S2->S4}`, and 4) `mov{CinderDB, S2->S3}`, `mov{CinderDB, S1->S4}`. The first two plans correspond to those shown in §4.1.

5 Limitations and Discussions

We discuss three high-level limitations of CloudCanary and potential ways to address them.

Quality of inputs. CloudCanary takes two types of inputs as given: a) dependencies, and b) failure probabilities; so it would be limited by the accuracy of the inputs (see §6.5 for a concrete example). For instance, an operator might not know that two upstream ISPs share the same undersea fiber, and that a fiber cut would bring down both networks; or an operator's estimate of the failure probabilities might not be perfectly accurate. In such cases, CloudCanary cannot automatically identify these inaccuracies. However, CloudCanary can benefit from advances in dependency collection systems or failure estimation algorithms: enhancement to CloudCanary's inputs always leads to improved utility.

Dependency granularity. CloudCanary is also limited by the dependency granularity of its data acquisition system; it currently cannot reason about more fine-grained dependencies such as configuration files. If a misconfigured component handles two upper-layer services differently, the current version of CloudCanary would not be able to identify that. This is somewhat akin to the previous limitation, and could benefit from a similar solution—*e.g.*, enhancing the fault graphs to capture configuration files.

Non-deterministic failures. The logic gates in CloudCanary's fault graph are deterministic, which assumes that if two services depend on a common component, the failure of the component would affect both services. This assumption does not capture well non-deterministic and/or partial failures, *e.g.*, when a bit flip in switch TCAM only affects a subset of services but not others. Modeling such behaviors might require extensions to the fault graph abstraction, which we leave as future work.

6 Evaluation

Our evaluation aims to answer three high-level questions: (1) How efficient and accurate is CloudCanary in identifying the risk groups? (2) How quickly can CloudCanary generate improvement plans? and (3) How well can CloudCanary shed light on failure risks in real-world traces?

Table 2: The configuration of our deployments.

	Deploy. A	Deploy. B	Deploy. C
# Switch ports	24	64	128
# Core routers	144	1,024	4,096
# Agg switches	288	2,048	8,192
# ToR switches	288	2,048	8,192
# Virtual machines	3,456	65,536	524,288
# Libraries/Microservices	4,492	79,824	638,592
Total # of components	8,668	150,480	1,183,360

Prototype implementation. We have developed a CloudCanary prototype using a mix of C++, Python, and open-source software libraries. Our system consists of three components: a) fault graph generator, b) SnapAudit, and c) DepBooster. The fault graph generator uses NSDMiner [56] and TS [24] to acquire network and software dependency data, and uses INDaaS [70] to parse and generate fault graphs. Our SnapAudit prototype uses a) a high-performance MinCostSAT solver, Maxino [6], for solving the Boolean formulas that encode the fault graphs, b) the Z3 solver [27] for DNF conversion, and c) a fault graph parser based on pyeda [7] to optimize the encoding and transformation of formulas. Our DepBooster prototype uses a scalable open-source SAT model counter, ApproxMC [2], to compute failure probabilities.

6.1 Experimental Setup

We have emulated a datacenter network with a Clos topology [59], and installed Apache Hadoop 3.2.0 and ZooKeeper 3.4.0 as the cloud services. In the performance experiments, we varied the service size from 8,668 to 1,183,360 software and network components using up to 524 k virtual machines, as shown in Table 2. We also used a real failure probability distribution trace for network devices in our experiments. All machines have an Intel Xeon E5-1620 v2 Quad Core HT 3.7 GHz CPU and 16 GB memory.

Baseline systems. Table 3 presents the three state-of-the-art auditing systems that we have used as the baseline to compare CloudCanary against. Among these systems, INDaaS [70] is more accurate than RepAudit [71] and reCloud [22], but the latter two are faster. This is because the minimal risk group algorithm in INDaaS relies on an exhaustive search, which can produce 100% accurate results but scales poorly. RepAudit and reCloud trade accuracy for efficiency: the former uses a simple MaxSAT solving that cannot guarantee that the identified risk groups are minimal[1], and the latter uses sampling for approximation, which may miss risk groups. Furthermore, we have included a fourth baseline that we call ProbINDaaS [70], which is a randomized version of INDaaS that also relies on sampling for efficiency. We note that reCloud uses a more advanced sampling algorithm (*i.e.*, dagger sampling) than ProbINDaaS (*i.e.*, Monte Carlo), and that reCloud can additionally provide the ability to generate a deployment from scratch to meet a reliability goal. Unlike all these baseline systems, CloudCanary can generate *improvement plans* based on the current deployment, and it performs *incremental auditing while preserving accuracy*. As shown later, CloudCanary achieves 100% accuracy while outperforming all baselines.

[1]MaxSAT solving means: given an SAT formula with weight one to each clause, find truth values for its variables that maximize the combined weight of the satisfied clauses.

Table 3: All evaluated systems and their comparisons.

System	Accurate?	Efficient?	Imp. Plans?
INDaaS [70]	✓	×	×
ProbINDaaS [70]	×	✓✗	×
reCloud [22]	×	✓✗	×
RepAudit [71]	✓✗	✓✗	×
CloudCanary	✓	✓	✓

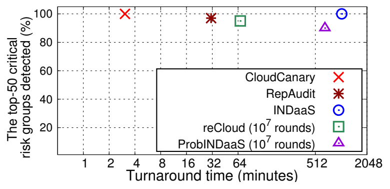
6.2 Performance: SnapAudit

We start by evaluating the performance of SnapAudit using the deployments in Table 2 (A: small, B: medium, C: large). For each deployment, we measured a) the time each system took to audit the service from scratch, and b) the time to audit an updated snapshot when 10% of the hosts and links have been affected. All audits asked for top-50 risk groups.

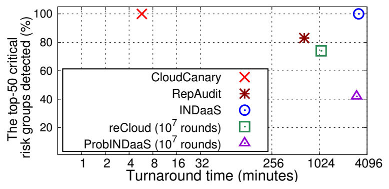
Efficiency. At service initiation, we observe that INDaaS is the slowest, taking up to $\sim$5811 minutes on the largest deployment. The two probabilistic approaches ProbINDaaS and reCloud (both with 10^7 sampling rounds) also perform poorly due to the large search space. RepAudit outperforms other baselines and is slightly ($\sim$1.3$\times$) faster than SnapAudit's FirstAudit. However, this is expected, because RepAudit only audits the overall fault graph, whereas FirstAudit audits *both* the overall fault graph *and* its subgraphs to create reusable results for subsequent audits.

We then updated the three deployments by randomly adding or removing 10% hosts and links, and ran the four auditing systems on the resulting deployments A′–C′. Figure 10 shows the turnaround time (X-axis) versus audit accuracy (Y-axis). As we can see, SnapAudit's IncAudit consistently outperforms INDaaS, ProbINDaas, reCloud, and RepAudit for all subsequent audits. On deployment B, IncAudit is faster than the second fastest system RepAudit by 200$\times$.

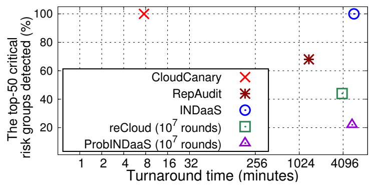
Accuracy. Moreover, SnapAudit always has an accuracy of 100% across deployments—the same with INDaaS—but RepAudit only has 96%, 83%, and 68% in deployment A'–C', respectively. ProbINDaaS and reCloud are even less accurate. Here, an inaccurate audit in RepAudit, reCloud, and ProbINDaaS means that a) some risk groups are missing from the output, and b) some risk groups generated by these systems are not minimal. For instance, SnapAudit outputs $\Sigma = \{\{A\},\{B\},\{C,D\}\}$ as the top-3 risk groups. An inaccurate system, however, may output $\Sigma' = \{\{A,E\},\{C,D,E\},\{C,D,F\}\}$, where $\{B\}$ is missing and the rest of the risk groups are not minimal. Therefore, even a low inaccuracy rate (*e.g.*, 100%−96%=4%) means that human operators need to *manually* inspect the results to identify re-



(a) Deployment A'.



(b) Deployment B'.



(c) Deployment C'.

Figure 10: Performance evaluation of CloudCanary, INDaaS, reCloud (with 10^7 rounds of sampling) and ProbINDaaS (with 10^7 rounds of sampling), and RepAudit in one of the update snapshots.

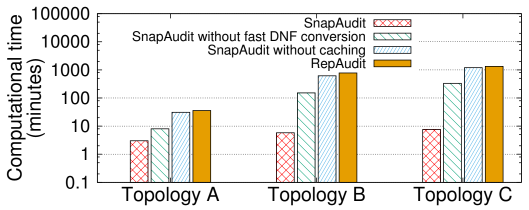


Figure 11: SnapAudit microbenchmarks.

dundancy and reason about the possibility of unidentified risk groups—a task that is time-consuming to perform at runtime.

Microbenchmarks. To further understand the performance improvements of the incremental auditing algorithm in SnapAudit, we have performed a set of microbenchmarks to break down the speedups. We used RepAudit as the baseline, as it performs faster than other systems and is closest to SnapAudit in its use of SAT solvers. For each of the deployments A'–C', we measured the execution times for four scenarios: a) SnapAudit with all optimizations turned on, b) SnapAudit without the fast DNF conversion, c) SnapAudit further without caching previous results, and d) RepAudit. Figure 11 shows that, without any optimization, SnapAudit performs very similarly with RepAudit; the slight speedup comes from the differences in the SAT formulations. The fast DNF conversion led to speedups of $2\times$–$40\times$, and reusing cached results led to speedups of $4\times$–$8\times$. These results demonstrate that the optimization techniques in SnapAudit can significantly accelerate incremental auditing.

Degrees of update. A third observation is that the time INDaaS, ProbINDaas, reCloud, and RepAudit took on each subsequent audit is roughly the same with that on their first audits, because they perform each audit from scratch. SnapAudit's IncAudit, on the other hand, is significantly faster on subsequent audits than its first audit.

To further evaluate how the degree of updates affects the auditing time of SnapAudit, we tested updates that affect 10%–50% of the components in deployment C, and used SnapAudit to audit these five updates. As shown in Figure 12, the turnaround time of CloudCanary increases roughly linearly with the update percentage. This is good news, because a complete overhaul of a deployed service is rare. Most updates only affect a small part of the service, and they can reap the benefits of CloudCanary easily. On the contrary, since RepAudit never used any incremental algorithm, the RepAudit performance in Figure 10 reflects the update that affects the majority of the deployment.

6.3 Performance: DepBooster

We now evaluate the performance of DepBooster for computing failure probabilities and generating improvement plans. For the first task, our baseline systems are INDaaS and RepAudit, both of which solve a Markov chain to obtain the probability. For the second task, our baseline systems are reCloud and RepAudit, although they are not designed to generate improvement plans directly.

Failure probability computation. Figure 13 shows the time DepBooster, reCloud (with 10^7 sampling rounds) and the baseline system (Markov chain computation) took to compute the failure probability of each deployment. For DepBooster, we set the precision per leaf node to be 10^{-4} (defined in §4.2) when adding virtual leaf nodes. As shown in Figure 13, DepBooster achieves a speedup of two to three orders of magnitude compared to reCloud and the baseline. On the largest deployment, DepBooster only took 2.5 minutes, whereas the baseline and reCloud took more than 10 hours. In terms of the failure probability precision, we found that DepBooster approximates the probability of the baseline system (which does not use any approximation) with an error of 10^{-3} for all tested deployments, whereas the error in reCloud is 10^{-2} for all tested deployments.

Improvement plan generation. Next, we evaluate the performance of DepBooster, using reCloud as the baseline. Each deployment hosted the service instances on 50% of the servers, and the query asked for improvement plans to reduce the failure probability to under 0.008 using the `mov` strategy. Figure 14 shows the results. We can see that DepBooster finished within 30 minutes across deployments, and outperforms reCloud and RepAudit by at least one order of magnitude. DepBooster can do better for two reasons. First, the model counter-based failure probability computation (§4.2) speeds up the result checking for each candidate solution. In fact, Figure 13 can also be looked as the microbenchmark evaluation for DepBooster, because the bottleneck operation of DepBooster is failure probability computation. Second, the pruning technique (§4.3) reduces the number of solutions searched; thus, we can observe few failure probability computations are needed.

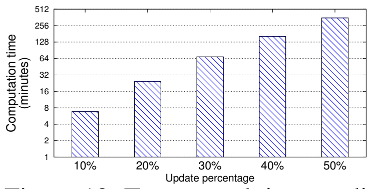


Figure 12: Turnaround time on different degrees of update.

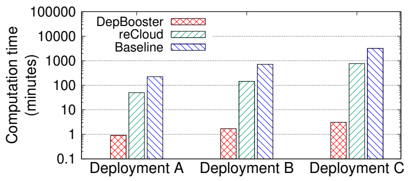


Figure 13: Performance: computing failure probability.

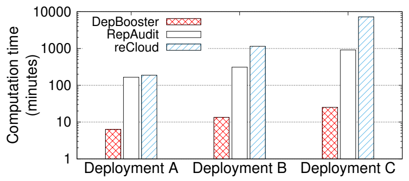


Figure 14: Performance: generating improvement plans.

6.4 Case Study

To better understand how the auditing (in)efficiency affects real-time updates, we have performed a case study that emulates a series of network and software updates. They consisted of 52 updates over the span of one week—we collected the update frequency and distribution from a large-scale cloud provider. We adapted six of these updates from realistic update scenarios [37, 48, 57] and from the Apache issue tracker [13]. All other updates were randomly generated and each of them affected 10% nodes in the deployment. This set of experiments was conducted over a deployment with 576 64-port core routers, 1,152 64-port aggregation switches, 1,152 64-port top-of-rack switches, and 27,648 servers.

Figure 15 shows the results for the first six updates, which we adapted from existing work.

- **Snapshot S0.** At service initialization, the operator set up the entire service, and performed an audit from scratch using the four auditing systems.
- **Snapshot S1: Small updates [48].** The first update changed 1% of network links.
- **Snapshot S2: Large updates [57].** The second update changed 20% servers and 20% links based on a network update trace, and it is designed as "pressure test".
- **Snapshots S3 and S4: Frequent updates [37].** The subsequent two updates occurred within a short interval of seven hours, designed as another pressure test.
- **Snapshots S5 and S6: Software version updates.** The final two updates were to software dependencies, where ZooKeeper was updated from version 3.4.0 to 3.4.6, and then to 3.4.8. They were designed to test the systems' ability to identify software-level risk groups.

Identifying risk groups. Figure 15 shows that existing systems are too inefficient for real-time auditing. INDaaS was only able to finish the auditing for S0 (at service initialization) and S6, but failed for all other updates, because its turnaround time exceeded the intervals between them. RepAudit and reCloud took roughly eight and sixteen hours per snapshot, and they finished S0, S2, and S6, which happened to be spaced out from their previous updates by more than sixteen hours, But they failed to finish for S3–S5, which came close to each other. Recall that, as explained in §6.2, RepAudit and reCloud achieve this speedup by trading accuracy for efficiency, so operators still need to manually reason about missing and non-minimal risk groups after audits.

CloudCanary achieves 100% accuracy in all tested cases, outputting the same results with INDaaS on scenarios where INDaaS was able to finish. However, for each real-time audit, it only took 4.7–6.5 minutes, outperforming INDaaS by 290×, reCloud by 250×, and RepAudit by 150×–200×. The only case where CloudCanary was slightly slower than RepAudit was at service initialization—when auditing S0, CloudCanary needs to audit *all subgraphs* in the fault graph from scratch.

Generating improvement plans. We then ran DEPBOOSTER to generate improvement plans for each snapshot. Since the strategies in CloudCanary do not involve changes to software components, we only evaluated S0–S4, where the reliability can be improved using CloudCanary's `mov` strategy. Our reliability goal was specified as $spec = \{rcg > 5 \wedge fp < 0.008\} \wedge \{mov\}$, and we assigned the failure probability of each switch or server to be 0.002 [36]. For S0–S4, CloudCanary generated improvement plans in 4.87, 2.32, 5.77, 7.12 and 6.29 minutes, respectively. In all cases, CloudCanary finished well before the next update arrived. Furthermore, in order to test the effectiveness of our improvement, we injected errors via a chaos-monkey-like way, randomly killing four components, because our constraints set $rcg > 5$. We observed that the improved deployments never failed.

Overall success rates: We now report results for all 52 update snapshots. Our metric is the *success rate* of a system, defined as $r = \frac{m}{n}$, where m is the number of updates for which the system finished on time, and n is the total number of updates. In other words, $m - n$ is the number of updates that cannot be handled due to an audit system's high turnaround time. Overall, INDaaS failed on almost all cases ($r = 1.92\%$) due to its inefficiency. reCloud and RepAudit are faster, but still only had a success rate of 3.85%. CloudCanary, on the other hand, achieved a success rate of 100%, finishing all audits with an average turnaround time of 5.23 minutes.

6.5 Identifying Real Risk Groups

Finally, we evaluate the usability of CloudCanary using a real-world update trace collected from a major service provider. The trace contains 300+ updates to its infrastructure, including software microservices, power sources, and network switches. The operators that executed these updates have already been trained with best practices for service reliability, but systematically understanding service dependencies is always a challenging task. For each update in this trace, we have used CloudCanary to identify the top-5 risk groups, and obtained

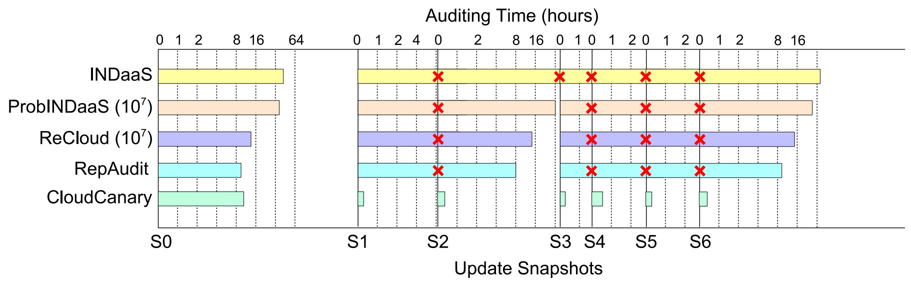


Figure 15: Results on a deployment running Hadoop 3.2.0 and ZooKeeper 3.4.0 in a Clos-topology datacenter with 27,648 hosts and 880 routers. S_i are service updates, which potentially lead to new risk groups.

feedback from the operators. Upon their request, the numbers below are presented as 50+, 10+, and so on, by rounding off their last digits. A key highlight here is that operators have confirmed that 50%+ of these risk groups were previously unknown to them, and that some of them actually caused service downtime in the past.

Microservices. We found 50+ risk groups in the microservice updates. The operators have confirmed that 96% of them could lead to correlated failures; the rest 4% are due to false positives of the dependency collection tool (see §5 for discussion on quality of inputs). One particularly risky example from the operators' feedback is an update that routes all requests to the same authentication service on a single machine. If this machine fails, this would lead to a major outage. The operators can fix this risk group by replicating the authentication service across multiple machines.

Power sources. We found 10+ risk groups in the power sources. Operators have confirmed that all of them could lead to correlated failures, and, in fact, 30%+ of them did trigger service downtime in the past. As a highlight, one of the updates assigned primary and backup power sources in the same cluster to serve several racks hosting a critical service. The cloud provider had experienced multiple hours of downtime due to a failure of these power sources.

Network. CloudCanary reported 30+ risk groups, including ToR/aggregation switches and shared fiber, all of which have been confirmed by the operators. As an example, we found that multiple data centers in the same city shared the same fiber bundles, which presents a risk of correlated failures. These risks can be prevented by adding redundant fiber bundles across different cities.

7 Related Work

Structural reliability auditing. The most relevant to CloudCanary are structural reliability auditing systems, INDaaS [70], reCloud [22], and RepAudit [71], which can construct fault graphs from dependency data and perform audits to prevent correlated failures. INDaaS and CloudCanary have higher accuracy than RepAudit and reCloud, because the latter two use approximate algorithms to trade accuracy for efficiency. Moreover, different from all existing work, CloudCanary is designed to perform *incremental auditing* over service updates.

Network/System verification. Failure prevention can also be achieved by formal analysis, such as configuration verification [16, 19, 31, 32, 50, 55, 60, 68], and synthesis/repair [29, 30, 48, 52, 65]. Some of these systems also use incremental verification for speedup when performing analysis [40, 43, 44]. Compared to these systems, CloudCanary has a very different goal—it aims at preventing correlated failures resulting from common dependencies—and also involves completely different algorithms as a result. On the contrary, network verification and synthesis systems primarily focus on reachability and performance properties, such as host-to-host connectivity. Similarly, software misconfiguration detection tools like PCheck [67] also focused on configuration logic, rather than failures caused by common dependencies.

Post-failure diagnostics. Many diagnostic systems [14, 15, 21, 23, 26, 28, 41, 42, 45, 56, 58] and provenance systems [66, 76, 77] have been proposed for failure troubleshooting. CloudCanary aims at a different goal from these efforts.

8 Conclusion

We have presented CloudCanary, a system that can perform real-time audits to prevent correlated failures in service updates. Our system can compute the risk groups in a service snapshot using cached results from previous audits, and it can generate improvement plans with increased reliability. It achieves this using a set of novel techniques, such as incremental auditing and network pruning. CloudCanary outperforms state-of-the-art systems by 200× and can generate improvement plans for large deployments within several minutes. Moreover, it can yield valuable insights over real-world traces from production environments.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Meg Walraed-Sullivan, for their insightful comments. We also thank Nikolaj Bjørner and Bryan Ford for their valuable feedback on the earlier version of this work. This work was partially supported by the National Science Foundation under Grant Numbers CCF-1715387, CCF-1553168, CCF-1302327, and CNS-1801884.

References

[1] Active Geo-Replication. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-active-geo-replication.

[2] ApproxMC. http://www.cs.rice.edu/CS/Verification/Projects/ApproxMC/.

[3] Correlated failures within EBS and EC2. https://aws.amazon.com/message/680342/.

[4] Cost of Data Center Outages. http://datacenterfrontier.com/white-paper/cost-data-center-outages/.

[5] Google: Gmail incident report. https://goo.gl/KY8mjp.

[6] Maxino: A fast MinCostSAT solver. https://github.com/alviano/aspino.

[7] Pyeda. https://github.com/cjdrake/pyeda.

[8] Rack Awareness. https://www.aerospike.com/docs/architecture/rack-aware.html.

[9] Rackspace Outage Nov 12th. https://goo.gl/J98iFz.

[10] Use Canary Tests to Test in Production. https://www.infoq.com/news/2013/03/canary-release-improve-quality.

[11] Walks in the neighborhood: Correlated failure in distributed systems. http://psteitz.blogspot.com/2011/10/correlated-failure-in-distributed.html.

[12] What I wish systems researchers would work on. http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html.

[13] ZooKeeper Issue Tracker. https://https://issues.apache.org/jira/projects/ZOOKEEPER.

[14] Bahl, P., Chandra, R., Greenberg, A. G., Kandula, S., Maltz, D. A., and Zhang, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2007).

[15] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for request extraction and workload modelling. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2004).

[16] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. A general approach to network configuration verification. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2017).

[17] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. Control plane compression. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2018).

[18] Bodík, P., Menache, I., Chowdhury, M., Mani, P., Maltz, D. A., and Stoica, I. Surviving failures in bandwidth-constrained datacenters. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2012).

[19] Canini, M., Jovanovic, V., Venzano, D., Novakovic, D., and Kostic, D. Online testing of federated and heterogeneous distributed systems. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2011).

[20] Chakraborty, S., Meel, K. S., and Vardi, M. Y. A scalable approximate model counter. In *19th International Conference on Principles and Practice of Constraint Programming (CP)* (Sept. 2013).

[21] Chen, M. Y., Accardi, A., Kiciman, E., Patterson, D. A., Fox, A., and Brewer, E. A. Path-based failure and evolution management. In *1st USENIX Symposium on Networked System Design and Implementation (NSDI)* (Mar. 2004).

[22] Chen, R., Akkus, I. E., Viswanath, B., Rimac, I., and Hilt, V. Towards reliable application deployment in the cloud. In *13th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)* (Dec. 2017).

[23] Chen, X., Zhang, M., Mao, Z. M., and Bahl, P. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2008).

[24] Chothia, Z., Liagouris, J., Dimitrova, D., and Roscoe, T. Online reconstruction of structural information from datacenter logs. In *12th European Conference on Computer Systems (EuroSys)* (Apr. 2017).

[25] Cidon, A., Rumble, S., Stutsman, R., Katti, S., Ousterhout, J., and Rosenblum, M. Copysets: Reducing the frequency of data loss in cloud storage. In *USENIX Annual Technical Conference (ATC)* (June 2013).

[26] Cohen, I., Chase, J. S., Goldszmidt, M., Kelly, T., and Symons, J. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2004).

[27] de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Mar. 2008).

[28] Dunagan, J., Harvey, N. J. A., Jones, M. B., Kostic, D., Theimer, M., and Wolman, A. FUSE: Lightweight guaranteed distributed failure notification. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2004).

[29] El-Hassany, A., Tsankov, P., Vanbever, L., and Vechev, M. T. Network-wide configuration synthesis. In *29th International Conference on Computer Aided Verification (CAV)* (July 2017).

[30] El-Hassany, A., Tsankov, P., Vanbever, L., and Vechev, M. T. NetComplete: Practical network-wide configuration synthesis with autocmpleteion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Apr. 2018).

[31] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., and Varghese, G. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2016).

[32] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (May 2015).

[33] FORD, B. Icebergs in the clouds: the *other* risks of cloud computing. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (June 2012).

[34] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2010).

[35] FU, Z., AND MALIK, S. Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search. In *International Conference on Computer-Aided Design (ICCAD)* (Nov. 2006).

[36] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2011).

[37] GOVINDAN, R., MINEI, I., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Evolve or die: High-availability design principles drawn from Google's network infrastructure. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2016).

[38] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *5th ACM Symposium on Cloud Computing (SoCC)* (Nov. 2014).

[39] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why does the cloud stop computing? Lessons from hundreds of service outages. In *7th ACM Symposium on Cloud Computing (SoCC)* (Oct. 2016).

[40] HORN, A., KHERADMAND, A., AND PRASAD, M. Delta-net: Real-time network verification using atoms. In *Proc. NSDI* (2017).

[41] KANDULA, S., KATABI, D., AND VASSEUR, J.-P. Shrink: A tool for failure diagnosis in IP networks. In *MineNet* (Aug. 2005).

[42] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2009).

[43] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *Proc. NSDI* (2013).

[44] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *Proc. NSDI* (2013).

[45] KOMPELLA, R. R., YATES, J., GREENBERG, A. G., AND SNOEREN, A. C. IP fault localization via risk modeling. In *2nd USENIX Symposium on Networked System Design and Implementation (NSDI)* (May 2005).

[46] KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell. 27*, 1 (1985), 97–109.

[47] LENERS, J. B., WU, H., HUNG, W., AGUILERA, M. K., AND WALFISH, M. Detecting failures in distributed systems with the Falcon spy network. In *23rd ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2011).

[48] LIU, H. H., WU, X., ZHANG, M., YUAN, L., WATTENHOFER, R., AND MALTZ, D. A. zUpdate: updating data center networks with zero loss. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2013).

[49] LIU, H. H., ZHU, Y., PADHYE, J., CAO, J., TALLAPRAGADA, S., LOPES, N. P., RYBALCHENKO, A., LU, G., AND YUAN, L. CrystalNet: Faithfully emulating large production networks. In *26th ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2017).

[50] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked System Design and Implementation (NSDI)* (May 2015).

[51] MAJDARA, A., AND WAKABAYASHI, T. Component-based modeling of systems for automated fault tree generation. *Reliability Engineering & System Safety 94*, 6 (2009), 1076–1086.

[52] MCCLURG, J., HOJJAT, H., CERNÝ, P., AND FOSTER, N. Efficient synthesis of network updates. In *36th ACM Conference on Programming Language Design and Implementation (PLDI)* (June 2015).

[53] MERKLE, R. C. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy (IEEE S&P)* (Apr. 1980).

[54] NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Subtleties in tolerating correlated failures in wide-area storage systems. In *3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)* (May 2006).

[55] PANDA, A., LAHAV, O., ARGYRAKI, K. J., SAGIV, M., AND SHENKER, S. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Mar. 2017).

[56] PEDDYCORD III, B., NING, P., AND JAJODIA, S. On the accurate identification of network service dependencies in distributed systems. In *26th Large Installation System Administration Conference (LISA)* (Dec. 2012).

[57] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for network update. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2012).

[58] REYNOLDS, P., KILLIAN, C. E., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *3rd Symposium on Networked Systems Design and Implementation (NSDI)* (May 2006).

[59] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2015).

[60] TIAN, B., ZHANG, X., ZHAI, E., LIU, H. H., YE, Q., WANG, C., WU, X., JI, Z., SANG, Y., ZHANG, M., YU, D., TIAN, C., ZHENG, H., AND ZHAO, B. Y. Safely and automatically updating in-network ACL configurations with intent language. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2019).

[61] Veeraraghavan, K., Meza, J., Michelson, S., Panneerselvam, S., Gyori, A., Chou, D., Margulis, S., Obenshain, D., Padmanabha, S., Shah, A., Song, Y. J., and Xu, T. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2018).

[62] Velev, M. N. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In *Asia South Pacific Design Automation (ASP-DAC)* (Jan. 2004).

[63] Vesely, W. E., Goldberg, F. F., Roberts, N. H., and Haasl, D. F. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, Jan. 1981.

[64] Wu, X., Turner, D., Chen, C.-C., Maltz, D. A., Yang, X., Yuan, L., and Zhang, M. NetPilot: Automating datacenter network failure mitigation. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2012).

[65] Wu, Y., Chen, A., Haeberlen, A., Zhou, W., and Loo, B. T. Automated bug removal for software-defined networks. In *14th USENIX Symposium on Networked System Design and Implementation (NSDI)* (Mar. 2017).

[66] Wu, Y., Zhao, M., Haeberlen, A., Zhou, W., and Loo, B. T. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2014).

[67] Xu, T., Jin, X., Huang, P., Zhou, Y., Lu, S., Jin, L., and Pasupathy, S. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2016).

[68] Yabandeh, M., Knezevic, N., Kostic, D., and Kuncak, V. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *6th USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)* (Apr. 2009).

[69] Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., and Pasupathy, S. SherLog: Error diagnosis by connecting clues from run-time logs. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Mar. 2010).

[70] Zhai, E., Chen, R., Wolinsky, D. I., and Ford, B. Heading off correlated failures through Independence-as-a-service. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2014).

[71] Zhai, E., Piskac, R., Gu, R., Lao, X., and Wang, X. An auditing language for preventing correlated failures in the cloud. In *32th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Oct. 2017).

[72] Zhai, E., Wolinsky, D. I., Xiao, H., Liu, H., Su, X., and Ford, B. Auditing the Structural Reliability of the Clouds. Tech. Rep. YALEU/DCS/TR-1479, Department of Computer Science, Yale University, 2013. Available at `http://cpsc.yale.edu/sites/default/files/files/tr1479.pdf`.

[73] Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., and Zhou, Y. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *26th ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2017).

[74] Zhao, X., Rodrigues, K., Luo, Y., Yuan, D., and Stumm, M. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2016).

[75] Zhao, X., Zhang, Y., Lion, D., Ullah, M. F., Luo, Y., Yuan, D., and Stumm, M. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2014).

[76] Zhou, W., Fei, Q., Narayan, A., Haeberlen, A., Loo, B. T., and Sherr, M. Secure network provenance. In *23rd ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2011).

[77] Zhou, W., Fei, Q., Sun, S., Tao, T., Haeberlen, A., Ives, Z. G., Loo, B. T., and Sherr, M. NetTrails: a declarative platform for maintaining and querying provenance in distributed systems. In *ACM International Conference on Management of Data (SIGMOD)* (June 2011).

Gryff: Unifying Consensus and Shared Registers

Matthew Burke
Cornell University

Audrey Cheng
Princeton University

Wyatt Lloyd
Princeton University

Abstract

Linearizability reduces the complexity of building correct applications. However, there is a tradeoff between using linearizability for geo-replicated storage and low tail latency. Traditional approaches use consensus to implement linearizable replicated state machines, but consensus is inefficient for workloads composed mostly of reads and writes.

We present the design, implementation, and evaluation of Gryff, a system that offers linearizability and low tail latency by unifying consensus with shared registers. Gryff introduces carstamps to correctly order reads and writes without incurring unnecessary constraints that are required when ordering stronger synchronization primitives. Our evaluation shows that Gryff's combination of an optimized shared register protocol with EPaxos allows it to provide lower service-level latency than EPaxos or MultiPaxos due to its much lower tail latency for reads.

1 Introduction

Large-scale web applications rely on replication to provide fault-tolerant storage. Increasingly, developers are turning to linearizable [32] storage systems because they reduce the complexity of implementing correct applications [2, 13, 17]. Recent systems from both academia [27, 35, 40, 52, 53, 57] and industry [6, 11, 14, 17, 23] demonstrate this trend.

Traditionally, linearizable storage systems for geo-replicated settings are built using state machine replication via consensus [33, 36, 37, 38, 45, 47, 50, 51]. These protocols are safe under the asynchronous network conditions that are common in wide-area networks. Furthermore, they provide the abstraction of a shared command log, which allows for the implementation of arbitrary deterministic state machines. Strong synchronization primitives, such as read-modify-write operations (rmws), can thus be used in applications built on top of these systems, further easing the programming burden on developers.

Linearizability for geo-replicated storage, however, comes with a tradeoff between strong guarantees and low latency. At least one communication delay between replicas is necessary to maintain a legal total order of operations [41], and in the wide-area, this communication incurs a considerable latency cost even in the best case. The tradeoff is starker for tail latency, where adverse conditions such as network delays, slow or failed replicas, and concurrent operations further delay responses to clients.

Tail latency is of particular importance for large-scale web applications, where end-user requests for high-level application objects fan-out into hundreds of sub-requests to storage services [18]. For example, when a user loads a page in a social networking service, an application server typically needs to invoke and wait for the completion of dozens of requests to replicas before returning the page to the client [2]. Only once the client receives the page can it begin loading additional assets and rendering the page. Thus, the median latency experienced by the end-user depends on the maximum of tens or hundreds of operations, which is dictated by the tail of the latency distribution.

Consensus protocols demonstrate the tradeoff between strong guarantees and low tail latency. Fundamentally, no protocol can solve consensus and guarantee termination in an asynchronous system with failures [24]. In practice, this impossibility result manifests as performance inefficiencies, such as serializing operations through a designated leader or delaying concurrent operations. In geo-replicated settings at scale, these inefficiencies impact tail latency.

In contrast, shared register protocols can implement linearizable shared registers, which support simple reads and writes, and guarantee termination in asynchronous systems with failures [5]. This translates to favorable tail latency for real protocols: shared register protocols are typically leaderless and often do not delay reads or writes, even if there are concurrent operations. The reads and writes provided by shared registers are the dominant types of operations in large-scale web applications [9]. Yet, shared registers are fundamentally too weak to directly implement strong synchronization primitives like rmws [31]. To resolve this tradeoff, the solution is to combine the strong synchronization provided by consensus with the favorable read/write tail latency of shared registers in a single protocol.

The idea of unifying consensus and shared registers is not new [8]. However, the only previous attempt of which we are aware is incorrect because it does not safely handle certain interleavings of operations. Our key insight is that protocol-level mechanisms for enforcing the interaction between rmws and reads/writes are difficult to reason about, which can lead to subtle safety violations. Instead, we argue the interaction be enforced at a deeper level, in the ordering mechanism itself, to simplify reasoning about correctness.

We introduce consensus-after-register timestamps, or *carstamps*, a novel ordering mechanism for distributed storage to leverage this insight. Carstamps allow writes and rmws to concurrently modify the same state without serializing through a leader or incurring additional round trips. Reads use carstamps to determine consistent values without interposing on concurrent updates.

Gryff is our system that implements this ordering mechanism to achieve unification.[1] It is the first such system to be proven correct, implemented, and empirically evaluated. Gryff combines a multi-writer variant [43] of the ABD [5] protocol for reads and writes with EPaxos [47] for rmws. In addition to the challenges associated with unifying these protocols, we introduce an optimization to further rein in tail latency by reducing the frequency of reads taking multiple wide-area round trips.

We implemented Gryff in the same framework as EPaxos [47] and MultiPaxos [36] and evaluated its performance in a geo-replicated setting. Our evaluation shows that Gryff reduces the tradeoff between linearizability and low tail latency for workloads representative of large-scale web applications [10, 16, 17]. For moderate contention workloads, Gryff reduces p99 read latency to $\sim$56% of EPaxos, but has $\sim 2x$ higher write latency. This tradeoff allows Gryff to reduce service-level p50 latency to $\sim$60% of EPaxos for large-scale web applications whose requests fan-out into many storage-level requests.

In summary, the contributions of this paper include:

- A novel ordering mechanism, carstamps, that enables efficient unification of consensus with shared registers. (§3)
- The Gryff design that combines a shared register protocol with EPaxos to provide reads, writes, and rmws. (§4, §5)
- The implementation and evaluation of Gryff, which demonstrates its latency improvements. (§6)

2 Consensus vs. Shared Registers

This section covers preliminaries and then compares and contrasts consensus and shared register protocols. It looks at the interfaces they support, the ordering constraints they impose, and the ordering mechanisms they use.

Model and Preliminaries. We study systems comprised of a set P of m processes that communicate with each other over point-to-point message channels. Processes may fail according to the *crash failure model*: a failed process ceases executing instructions and its failure is not detectable by other processes. The system is *asynchronous* such that there is no upper bound on the time it takes for a message to be delivered and there is no bound on the relative speeds at which processes execute instructions.

Linearizability is a correctness condition for a concurrent object that requires (a) operations invoked by processes accessing the object appear to execute in some total order that is consistent with the semantics of the object (i.e., that is *legal*) and (b) the total order is consistent with the order that operations happened in real time [32]. Linearizability is a *local* property, meaning it holds for a collection of objects if and only if it holds for each individual object.

[1]A gryffin is a mythological hybrid creature that combines the power of a lion with the speed of an eagle.

For the remainder of this text, we consider linearizable replication of a single object by omitting object identifiers; it is straightforward to compose instances of such a system to obtain a linearizable multi-object system.

2.1 State Machines and Consensus

State machine replication is the canonical approach to implementing fault-tolerant services [56]. It provides a fault-tolerant *state machine* that exposes the following interface:

- $\textsc{Command}(c(\cdot))$: atomically applies a deterministic computation $c(\cdot)$ to the state machine and returns any outputs

Each command can include zero or more arguments, read local state, perform deterministic computation, and produce output. The state machine approach applies these commands one by one starting from the same initial state to move replicas through identical states. Thus, if some replicas fail, the remaining replicas still have the state and can continue to provide the service.

Applying commands in the same order on all replicas requires an ordering mechanism that is *stable*, i.e., a replica knows when a command's position is fixed and it will never receive an earlier command [56]. In asynchronous systems where processes can fail, consensus protocols [33, 36, 37, 38, 45, 47, 50, 51] are used to agree on this stable ordering.

Figure 1a shows the stable ordering provided by consensus protocols for state machine replication. Commands are assigned positions in a log and a command becomes stable once there are no empty slots preceding its own in the log.

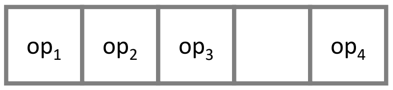
2.2 Shared Registers and Their Protocols

A *shared register* has the following interface:

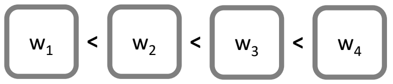
- $\textsc{Read}()$: returns the value of the register
- $\textsc{Write}(v)$: updates the value of the register to v

Shared registers provide a simple interface with read and write operations. They are less general than state machines as they provably cannot be used to implement consensus [31]. Shared register protocols replicate shared registers across multiple processes for fault tolerance [5, 22, 43].

Shared register protocols provide a linearizable ordering of operations. That ordering does not have to be stable, however, because each write operation fully defines the state of the object. Thus, a replica can safely apply a write w_4 even if it does not know about earlier writes. If an earlier write w_3 ever does arrive, the replica simply ignores that write because it already has the resulting state from applying w_3 and then w_4. Figure 1b shows shared register ordering where there is a total order of all writes (denoted by $<$) without stability.

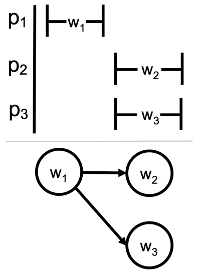


(a) Ordering in consensus protocols. Operations op_1, op_2, and op_3 are stable, but op_4 is not.

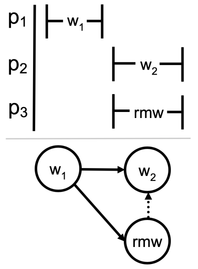


(b) Ordering in shared register protocols. No writes are stable.

Figure 1: Comparison of ordering in consensus and shared register protocols. Shared register protocols provide an unstable ordering where new writes can be inserted between writes that have already completed.



(a) w_2 and w_3 may be arbitrarily ordered.



(b) if rmw reads w_1, it must be before w_2.

Figure 2: Solid arrows are real time ordering constraints. Dashed arrows are operation semantic constraints.

2.3 Shared Objects and Their Ordering

A *shared object* exposes the following interface:

- READ(): returns the value of the object
- WRITE(v): updates the value of the object to v
- RMW($f(\cdot)$): atomically reads the value v, updates the value to $f(v)$, and returns v

The abstraction of a shared object captures an intuitive programming model that is used in real-world systems [12, 15, 23, 44, 54, 55]. Most operations read or write data, but rmws support stronger primitives to synchronize concurrent accesses to data. For example, a conditional write can be implemented with a rmw by using a function $f(\cdot)$ that returns the new value to be written only if some condition is met.

Shared objects and state machines are equivalent in that an instance of one can be used to implement the other [31]. However, the difference is that shared objects expose a more restrictive interface for directly reading and writing state, as do shared registers. These simpler operations can be implemented more efficiently because their semantics impose fewer ordering constraints.

Yet, neither the stable ordering of state machine replication nor the unstable total ordering of shared register protocols is a good fit for shared objects. A stable order, on the one hand, over constrains how reads and writes are ordered and results in less efficient protocols. On the other hand, an unstable total order under constrains how rmws are ordered and results in an incorrect protocol.

Figure 2 demonstrates these different constraints. Consider the execution in Figure 2a where two processes, p_2 and p_3, write concurrently. Linearizability stipulates that w_2 and w_3 be ordered after w_1 because they are invoked after w_1 completes in real time. However, there is no stipulation for how w_2 and w_3 are ordered with respect to each other because the result of a write does not depend on preceding operations. Both $w_1 \to w_2 \to w_3$ and $w_1 \to w_3 \to w_2$ are valid.

Now consider the execution in Figure 2b involving a rmw. Process p_2 writes while p_3 concurrently executes a rmw. The *base update* of a rmw is the operation that writes the value that the rmw reads. Assume that w_1 is the base update of *rmw*. Then, not only does *rmw* need to be ordered after w_1, but no other write may be ordered between w_1 and *rmw*. This additional constraint ensures legality because the semantics of a rmw requires that it must appear to atomically read and update the object based on the value read. Thus, only $w_1 \to rmw \to w_2$ is a valid order.

3 Carstamps for Correct Ordering

Consensus-after-register timestamps, or *carstamps*, precisely capture the ordering constraints of shared objects. They provide the necessary stable order for rmws and the more efficient unstable order for reads and writes. This section describes the requirements of a precise ordering mechanism for shared objects and then describes carstamps.

3.1 Precise Ordering for Shared Objects

An ordering mechanism is an injective function $g : X \to Y$ from a set X of writes and rmws to a totally ordered set $(Y, <_Y)$. A mechanism g produces a total order $<_g$ on X: for all $x_1, x_2 \in X$, $x_1 <_g x_2$ if and only if $g(x_1) <_Y g(x_2)$.

Typically, replication protocols augment an ordering mechanism with protocol-level logic to enforce real time and legality constraints on the total order given by the ordering mechanism to provide linearizability. While the logic for enforcing real time constraints is often straightforward, legality constraints can be more complex.

Protocol-level Legality. For example, consider the Active Quorum Systems (AQS) protocol [7, 8]. AQS is the only prior protocol of which we are aware that attempts to combine consensus and shared registers and it does so with an unstable ordering mechanism. This allows for executions where a rmw *rmw* with base update u is ordered such that there exists a $y \in Y$ with $g(u) <_Y y <_Y g(rmw)$. This can result in an illegal total order when a write w is concurrent with *rmw* because w may be assigned $g(w) = y$. AQS contains no logic at the protocol-level to prevent this subtle scenario. We discuss such an execution in detail in Appendix C and describe how there does not exist a linearizable order of all operations.

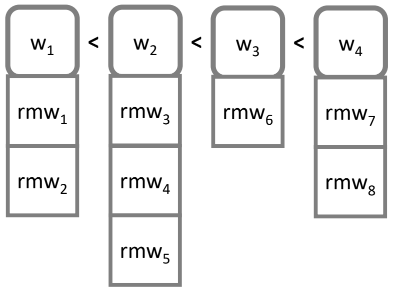


Figure 3: Unified ordering provided by carstamps for writes and rmws. Writes are unstably ordered while rmws are stably ordered with their base updates.

Ordering-level Legality. Our key insight is that the legality constraints of linearizability can be encoded in the ordering mechanism itself. An ordering mechanism that does this must ensure that for all $rmw \in X$ such that u is the base update of rmw, $g(u) <_Y g(rmw)$ and $g(u)$ is a *cover* of $g(rmw)$. This means that there is no $y \in Y$ such that $g(u) <_Y y <_Y g(rmw)$. With such an ordering mechanism, there is no need for protocol-level logic to prevent other writes in X from being assigned an illegal position in the total order between $g(u)$ and $g(rmw)$.

3.2 Carstamps

Our solution which leverages this insight is called *carstamps*. A carstamp is a triple $cs = (ts, id, rmwc)$ with three fields: a logical timestamp ts, a process identifier id, and a rmw counter $rmwc$. The logical timestamp and process identifier can be used by a write protocol to form an unstable order of writes. A rmw rmw with base update u whose carstamp is cs_u is assigned a carstamp $cs_{rmw} = (cs_u.ts, cs_u.id, cs_u.rmwc + 1)$. The fields encode ordering constraints between operations via a lexicographical comparison such that $cs_1 < cs_2$ if and only if $cs_1.ts < cs_2.ts$ or $cs_1.ts = cs_2.ts$ and $cs_1.id < cs_2.id$ or $cs_1.ts = cs_2.ts$ and $cs_1.id = cs_2.id$ and $cs_1.rmwc < cs_2.rmwc$.

By incrementing the lowest order field of the carstamp, each carstamp assigned to a base update of a rmw is guaranteed to cover its rmw. This stable ordering of rmws with their base updates is visualized in Figure 3. Writes are assigned to carstamps in the first row as part of an increasing unstable order. RMWs are assigned to carstamps in the column to which their base update belongs immediately below their base update.

Consider the example from Figure 2b and assume that w_1 is assigned carstamp $cs_{w_1} = (1,1,0)$ by p_1. Then, since rmw reads w_1, it will be assigned carstamp $cs_{rmw} = (1,1,1)$. Based on the lexicographical ordering of carstamps, there does not exist a carstamp cs such that $cs_{w_1} < cs < cs_{rmw}$, so w_2 cannot be arbitrarily re-ordered between w_1 and rmw.

4 Gryff Protocol

Gryff unifies shared registers with consensus using carstamps. It implements a linearizable shared object (§2) that tolerates the failure of up to f out of $n = 2f + 1$ replicas. We divide its description into three components. First, we provide additional background including the shared register protocol and consensus protocol upon which its read, write, and rmw protocols are built (§4.1). Second, we describe how Gryff adapts these protocols with carstamps (§4.2,§4.3). Third, we describe an optimization to the base Gryff protocol that improves read latency in geo-replicated settings (§5).

In addition, in Appendix B we prove Gryff implements a shared object with linearizability. Appendix B also proves read/write wait-freedom—every read or write invoked by a correct process eventually completes—and rmw wait-freedom with partial synchrony—if there is a point in time after which the system is synchronous, every rmw invoked by a correct process eventually completes.

4.1 Background

Section 2 provides background on our model, linearizability, and state machines and shared registers in general. This subsection adds useful definitions and then describes the two specific protocols that Gryff adapts, a multi-writer variant [43] of ABD [5] and EPaxos [47].

Definitions. A subset of processes $R \subseteq P$ are *replicas* that store the value of the object. We assume reliable message delivery, which can be implemented on top of unreliable message channels via retransmission and deduplication.

Replicas are often deployed across a wide-area network such that inter-replica message delivery latency is on the order of tens of milliseconds. This is commonly done so that replica or network failures correlated by geographic region do not immediately cause the system to become unavailable. We say that a process p is *co-located* with a replica r if the message delivery latency between p and r is much less than the minimum inter-replica latency. Client processes running applications are typically co-located with a single replica, for example, within the same datacenter.

A *quorum system* $\mathcal{Q} \subseteq \mathcal{P}(R)$ over R is a set of subsets of R with the *quorum intersection property*: for all $Q_1, Q_2 \in \mathcal{Q}$, $Q_1 \cap Q_2 \neq \emptyset$. We use *quorum* both to mean a set of replicas in a particular quorum system and the size of such a set. Gryff can use any quorum system, but for liveness with up to f replica failures, we assume the use of the majority quorum system $\mathcal{Q}_{maj}$ such that $\forall Q \in \mathcal{Q}_{maj}. |Q| = f + 1$.

A *coordinator* is a process that executes a read, write, or rmw protocol when it receives such an operation from an application. In shared register protocols, the coordinators are typically the client processes on which the application is running. In consensus protocols, the coordinators are typically one of the replicas to which client processes forward their requests. We assume all processes possess a unique *identifier* that can be used when coordinating an operation to distinguish the coordinator from other processes.

Multi-Writer ABD. The multi-writer variant [43] of ABD [5] is a shared register protocol that requires two phases for both reads and writes. To provide a linearizable order of reads and writes, it associates a *tag* $t = (ts, id)$ with each write where *ts* is a logical timestamp and *id* is the identifier of the coordinator. Writes are ordered lexicographically by their tags. Each replica stores a value v and an associated tag t.

Reads and writes have two phases. A read begins with the coordinator reading the current tag and value from a quorum. Once it receives these, it determines the value that will be returned by the read by choosing the value associated with the maximum tag from the tags returned in the quorum. Then, the coordinator propagates this maximum tag and value to a quorum and waits for acknowledgments. We say that a replica *applies* a value v' and tag t' when it overwrites its v and t with v' and t' if $t' > t$. After a replica receives the propagated tag and value, it applies them and sends an acknowledgment to the coordinator.

A coordinator for a write follows a similar two-phase protocol, except instead of propagating the maximum tag t_{max} and associated value received in the first phase, it generates a new tag $t = (t_{max}.ts + 1, id)$ to associate with the value to be written where *id* is the identifier of the coordinator. In the second phase, the coordinator propagates this new tag and value to a quorum and waits for acknowledgments.

EPaxos EPaxos [47] is a consensus protocol that provides optimal commit latency in the wide-area. It has three phases in failure-free executions: PreAccept, Accept, and Commit. If a command commits on the *fast path*, the coordinator returns to the client after the PreAccept phase and skips the Accept phase. Otherwise, the command commits on the *slow path* after the Accept phase. Commands that do not read state complete at the beginning of the Commit phase; commands that do read state complete after a single replica, typically the coordinator, executes the command to obtain the returned state. The purpose of the PreAccept and Accept phases is to establish the *dependencies* for a command, or the set of commands that must be executed before the current command. The purpose of the Commit phase is for the coordinator to notify the other replicas of the agreed-upon dependencies.

PreAccept phase. The coordinator of a command constructs the preliminary dependency set consisting of all other commands of which the coordinator is aware that interfere (i.e., access the same state machine state) with it. It sends the command and its dependencies to a fast quorum of replicas. When replicas receive the proposed dependencies, they update them with any interfering commands of which they are aware that are not already in the set and respond to the coordinator with the possibly updated dependencies. If the leader receives a fast quorum of responses that all contain the same dependencies, it proceeds to the Commit phase.

v - value of shared object
cs - carstamp of shared object
prev - value and carstamp generated by the previously executed rmw
i - next unused instance number
cmds - two-dimensional array of instances indexed by replica id and instance number each containing:

- *cmd* - command to be executed
- *deps* - instances whose commands must be executed before this one
- *seq* - approximate sequence number of command used to break cycles in dependency graph
- *base* - possible base update for rmw
- *status* - status of instance

Figure 4: State at each replica.

Accept phase. Otherwise, the coordinator continues to the Accept phase where it builds the final dependencies for the command by taking the union of all the dependencies that it received in the PreAccept phase. It sends these to a quorum and waits for a quorum of acknowledgments before committing. Regardless of whether the command is committed after the first or second phase, once it is committed, a quorum store the same dependency set for the command.

Execution. Dependency sets for distinct commands define a dependency graph over all interfering commands. The EPaxos execution algorithm, separate from the commit protocol, executes all commands in the deterministic order specified by the graph. Cycles may exist in the graph, in which case a total order is determined by a secondary attribute called an approximate sequence number. We refer the reader to the EPaxos paper for more details [47].

4.2 Read & Write Protocols

The read and write protocols are based on multi-writer ABD. Figure 4 summarizes the state that is maintained at each replica. Algorithms 1 and 2 show the pseudocode for the coordinators and replicas. The key difference from multi-writer ABD is that replicas maintain a carstamp associated with the current value of the shared object instead of a tag so that rmws are properly ordered with respect to reads and writes.

Reads. We make the same observation as Georgiou et al. [26] that the second phase in the read protocol of multi-writer ABD is redundant when a quorum already store the value and associated carstamp chosen in the first phase. In such cases, the coordinator may immediately complete the read (Line 6 of Algorithm 1). Otherwise, it continues as normal to the second phase in order to propagate the observed value and carstamp to a quorum.

Writes. When generating a carstamp after the first phase of a write, the coordinator chooses the *ts* and *id* fields as

Algorithm 1: Read and write coordinator protocols.

```
procedure Coordinator::READ() at $p \in P$
    send $Read1$ to all $r \in R$
    wait to receive $Read1Reply(v_r, cs_r)$ from all $r \in Q \in \mathcal{Q}$
    $cs_{max} \leftarrow \max_{r \in Q} cs_r$
    $v \leftarrow v_r : cs_r = cs_{max}$
    if $\forall r \in Q : cs_r = cs_{max}$ then
        return $v$
    send $Read2(v, cs_{max})$ to all $r \in R$
    wait to receive $Read2Reply$ from all $r \in Q' \in \mathcal{Q}$
    return $v$
procedure Coordinator::WRITE($v$) at $p \in P$
    send $Write1$ to all $r \in R$
    wait to receive $Write1Reply(cs_r)$ from all $r \in Q \in \mathcal{Q}$
    $cs_{max} \leftarrow \max_{r \in Q} cs_r$
    $cs \leftarrow (cs_{max}.ts + 1, id, 0)$
    send $Write2(v, cs)$ to all $r \in R$
    wait to receive $Write2Reply$ from all $r \in Q' \in \mathcal{Q}$
```

Algorithm 2: Read and write replica protocols.

```
when replica $r \in R$ receives a message $m$ from $p \in P$ do
    case $m = Read1$ do
        send $Read1Reply(v, cs)$ to $p$
    case $m = Read2(v', cs')$ do
        APPLY($v', cs'$)
        send $Read2Reply$ to $p$
    case $m = Write1$ do
        send $Write1Reply(cs)$ to $p$
    case $m = Write2(v', cs')$ do
        APPLY($v', cs'$)
        send $Write2Reply$ to $p$
procedure Replica::APPLY($v', cs'$)
    if $cs' > cs$ then
        $cs \leftarrow cs'$
        $v \leftarrow v'$
```

in multi-writer ABD. The *rmwc* field is reset to 0 (Line 15 of Algorithm 1). While not strictly necessary, this curbs the growth of the *rmwc* field in practical implementations.

4.3 Read-Modify-Write Protocol

Gryff's rmw protocol uses EPaxos to stably order rmws as commands in the dependency graph. Figure 4 summarizes the replica state. Algorithms 3 and 4 show the pseudocode for a rmw coordinator and replica message handling excluding the recovery procedure. Appendix B includes the pseudocode for the recovery procedure. The highlighted portions of the pseudocode show the changes from canonical EPaxos. We denote by I_{cmd} the set of commands of which the local replica is aware that interfere with *cmd*.

We make three high-level modifications to canonical EPaxos in order to unify its stable ordering with the unstable ordering of Gryff's read and write protocols.

1. A base update attribute, *base*, is decided by the replicas during the same process that establishes the dependencies and the approximate sequence number for a rmw.
2. A rmw completes after a quorum execute it.
3. When a rmw executes, it chooses its base update from between its *base* attribute and the result of the previously executed rmw *prev*. The result of the executed rmw is applied to the value and carstamp of the executing replica.

The first change adapts EPaxos to work with the unstable order of writes by fixing the write upon which it will operate. The second change adapts it to work with reads that bypass its execution protocol and directly read state. The third change ensures that concurrent rmws that choose the same initial base update are stably ordered using the ordering and execution protocols of EPaxos. We next discuss each of these changes in more detail.

Base Attribute. The *base* attribute associated with a rmw represents a possible base update on which the rmw will execute. Initially, the coordinator sets this to what it believes are the current value and carstamp of the shared object (Line 6 of Algorithm 3). When a replica receives a *PreAccept* message, it merges what it believes is the correct base update with the base update proposed by the coordinator (Line 5 of Algorithm 4). The fast path condition remains essentially unchanged: the coordinator commits the command if it receives *PreAcceptOK* responses from a fast quorum indicating that all replicas in the quorum agree on the attributes for the command. Otherwise, the coordinator merges all attributes it has received in the PreAccept phase and sends out the final attributes in the Accept phase.

Quorum Execute. In canonical EPaxos, a rmw completes after a single replica executes it because reads are executed through the same consensus protocol. Since Gryff's read protocol circumvents consensus and reads the state of the shared object directly from a quorum, a rmw must be executed at a quorum so that it is visible to reads that come after it in real time. This guarantees the rmw will be visible to future reads by the quorum intersection property.

Execution. The algorithm for determining the execution order of commands is unchanged from canonical EPaxos. The EXECUTE procedure in Algorithm 4 is called when a rmw *rmw* in the dependency graph committed at position (i, j) in the *cmds* array is ready to be executed.

In the procedure, the final base update for *rmw* is chosen to be the value and carstamp pair with the larger carstamp

Algorithm 3: RMW coordinator protocol.

```
procedure Coordinator::RMW($f(\cdot)$) at $c \in R$
    PreAccept Phase:
    $i \leftarrow i+1$
    $cmd \leftarrow f(\cdot)$
    $seq \leftarrow 1+\max(\{cmds[j][k].seq | (j,k) \in I_{cmd}\} \cup \{0\})$
    $deps \leftarrow I_{cmd}$
    $base \leftarrow (v,cs)$
    $cmds[id][i] \leftarrow (cmd, seq, deps, base,$ pre-accepted$)$
    send $PreAccept(cmd, seq, deps, base, id, i)$ to all $r \in F \setminus \{c\}$ where $F \in \mathscr{F}$
    wait to receive $PreAcceptOK(seq'_r, deps'_r, base'_r)$ from all $r \in F \setminus \{c\}$
    if $\forall r_1, r_2 \in F \setminus \{c\} : seq'_{r_1} = seq'_{r_2} \wedge deps'_{r_1} = deps'_{r_2} \wedge base'_{r_1} = base'_{r_2}$ then
        $deps, seq, base \leftarrow deps'_r, seq'_r, base'_r : r \in F \setminus \{c\}$
        goto Commit Phase
    Accept Phase:
    $deps \leftarrow \cup_{r \in F} deps_r$
    $seq \leftarrow \max_{r \in F} seq_r$
    $base \leftarrow base_r : \forall r' \in F.base_r.cs \geq base_{r'}.cs$
    $cmds[id][i] \leftarrow (cmd, seq, deps, base,$ accepted$)$
    send $Accept(cmd, seq, deps, base, id, i)$ to all $r \in Q \setminus \{c\}$ where $Q \in \mathscr{Q}$
    wait to receive $AcceptOK$ from all $r \in Q \setminus \{c\}$
    Commit Phase:
    $cmds[id][i] \leftarrow (cmd, seq, deps, base,$ committed$)$
    send $Commit(cmd, seq, deps, base, id, i)$ to all $r \in R \setminus \{c\}$
    wait to receive $Executed(v)$ from all $r \in Q' \in \mathscr{Q}$
    return $v$
```

Algorithm 4: RMW replica protocol.

```
when replica $r \in R$ receives a message $m$ from $c \in R$ do
    case $m = PreAccept(cmd, seq, deps, base, id_c, i)$ do
        $seq' \leftarrow \max(\{seq\} \cup \{1+cmds[j][k].seq | (j,k) \in I_{cmd}\}$
        $deps' \leftarrow deps \cup I_{cmd}$
        $base' \leftarrow$ if $cs > base.cs$ then $(v,cs)$ else $base$
        $cmds[id_c][i] \leftarrow (cmd, seq', deps', base',$ pre-accepted$)$
        send $PreAcceptOK(seq', deps', base')$ to $c$
    case $m = Accept(cmd, seq, deps, base, id_c, i)$ do
        $cmds[id_c][i] \leftarrow (cmd, seq', deps', base',$ accepted$)$
        send $AcceptOK$ to $c$
    case $m = Commit(cmd, seq, deps, base, id_c, i)$ do
        $cmds[id_c][i] \leftarrow (cmd, seq', deps', base',$ committed$)$

procedure Replica::EXECUTE$(j,k)$
    $base \leftarrow cmds[j][k].base$
    if $cmds[j][k].base.cs < prev.cs$ then
        $base \leftarrow prev$
    $v' \leftarrow cmds[j][k].cmd(base.v)$
    $cs' \leftarrow (base.cs.ts, base.cs.id, base.cs.rmwc+1)$
    $prev \leftarrow (v', cs')$
    APPLY$(v', cs')$
    send $Executed(base.v)$ to replica $j$
```

between the result *prev* of the previously executed rmw and the *base* attribute of *rmw* (Line 15 of Algorithm 4). The *prev* variable is the most recent state of the shared object produced by the execution of a rmw whereas the *base* attribute is the most recent state of the shared object that the coordinator observed after *rmw* was invoked. In the absence of concurrent updates, these states are equivalent, so it is safe for the *rmw* to choose the state as the base update.

However, when rmws are concurrent, *prev* may be more recent than the *base* attribute of *rmw* because concurrent rmws were ordered and executed before *rmw*. In such cases, *rmw* must remain consistent with the stable order of rmws provided by EPaxos by executing on the most recent state.

The resulting value and carstamp of *rmw* are decided by executing the modify function $f(\cdot)$ on the value of the base update and incrementing the *rmwc* of the carstamp of the chosen base update. The replica finishes by applying the new value and carstamp and notifying the coordinator that the rmw has been executed.

5 Proxying Reads

The base Gryff read protocol, as described in the previous section, provides reads with single round-trip time latency from the coordinator to the nearest quorum including itself (1 RTT) when there are no concurrent updates. Otherwise, reads have at most 2 RTT latency. We discuss how read latency can be further improved in deployments across wide-area networks.

Because the round-trip time to the replica that is co-located with a client process is negligible relative to the inter-replica latency, replicas can coordinate reads for their co-located clients and utilize their local state in the read coordinator protocol to terminate after 1 RTT more often. When using this optimization, we say that the coordinating replica is a *proxy* for the client process's read.

Propagating Extra Data in Read Phase 1. The proxy includes in the *Read1* messages its current value *v* and carstamp *cs*. Upon receiving a *Read1* message with this additional information, a replica applies the value and carstamp before returning its current value and carstamp. This has the effect of ensuring every replica that receives the *Read1* messages will have a carstamp (and associated value) at least as large as the carstamp at the proxy when the read was invoked.

When this is the most recent carstamp for the shared object, the read is guaranteed to terminate after 1 RTT. This is because every *Read1Reply* that the coordinator receives will contain this most recent carstamp and associated value.

Updating the Proxy's Data. The proxy also applies the values and carstamps that it receives in *Read1Reply* messages as it receives them and before it makes the decision of whether or not to complete the read after the first phase. If every reply contains the same carstamp, then the read completes after 1 RTT even if the carstamp at the proxy when the read was invoked is smaller than the carstamp contained in every reply.

Given our assumption that each quorum contains $f+1$ replicas, these two modifications ensure that reads coordinated by a proxy r only take 2 RTT during normal operation when there is a concurrent update that arrives at the f nearest replicas to r in an order that interleaves with the *Read1* messages from r. Algorithm 7 in Appendix B describes the read proxy changes to base Gryff in pseudocode. Appendix B also contains a brief argument for why the read proxy optimization maintains the correctness of base Gryff.

Always Fast Reads When $n = 3$. This optimization increases the likelihood that a read completes in 1 RTT because the proxy replica is privy to more information—i.e., the number of replicas that contain the same value and carstamp—than a client process. Moreover, it allows Gryff to always provide 1 RTT reads when $n = 3$ since the proxy and any single other replica comprise a quorum. This optimization is, in some sense, the dual of the optimization that EPaxos [47] uses to always provide 1 RTT writes when $n = 3$. In both cases, the coordinator and the other replica in the quorum adopt each other's state so that the quorum always has the same state at the end of the first phase.

6 Evaluation

Gryff unifies consensus with shared registers to avoid the overhead of consensus for reads and writes. To quantify the benefits and drawbacks of this approach for storing data in geo-replicated, large-scale web applications, we ask:

- Do Gryff's shared register read and write protocols reduce read tail latency relative to the state-of-the-art? (§6.3)
- How do the read/write/rmw latency and throughput of Gryff compare to state-of-the-art protocols? (§6.4,§6.5)
- Does Gryff improve the median service-level latency for large scale web applications? (§6.6)

We find that, for workloads with moderate contention, Gryff reduces p99 read latency to ∼56% of EPaxos, but has ∼2x higher write latency. This tradeoff allows Gryff to reduce service-level p50 latency to ∼60% of EPaxos for large-scale web applications whose requests fan-out into many storage-level requests. Gryff and EPaxos each achieve a slightly higher maximum throughput than MultiPaxos due to their leaderless structure.

	CA	VA	IR	OR	JP
CA	0.2				
VA	72.0	0.2			
IR	151.0	88.0	0.2		
OR	59.0	93.0	145.0	0.2	
JP	113.0	162.0	220.0	121.0	0.2

Figure 5: Round trip latencies in ms between nodes in emulated geographic regions.

6.1 Baselines and Implementation

We evaluate Gryff against MultiPaxos and EPaxos. MultiPaxos [36], VR [50], Raft [51] and other protocols with leader-based architectures are used in commercial systems to provide linearizable replicated storage [14, 17, 23, 52]. While leader-based protocols have drawbacks in geo-replicated settings, their extensive use in real systems provides a practical measuring stick. EPaxos [47] is the state-of-the-art for geo-replicated storage.

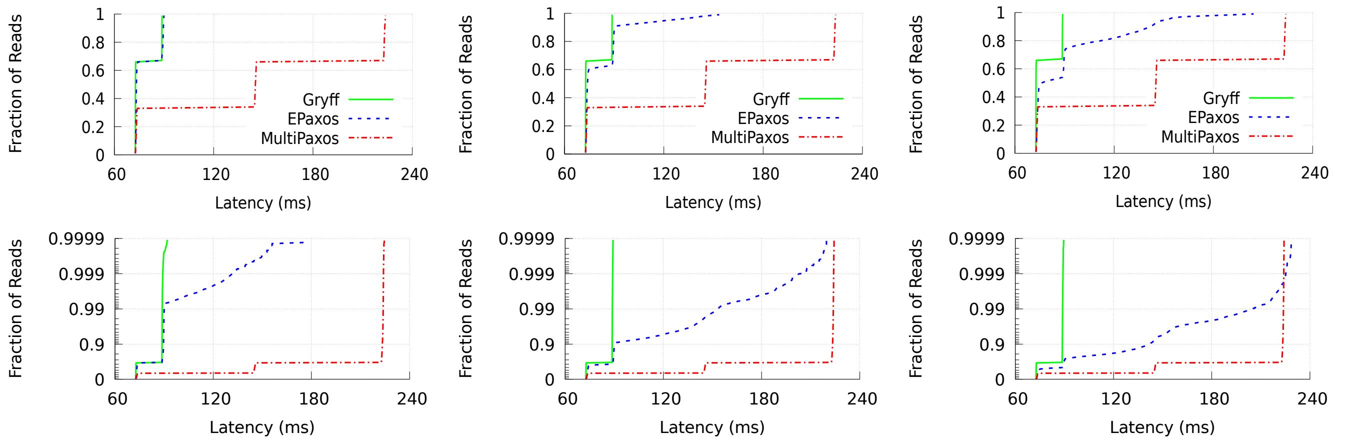
We implemented Gryff in Go using the framework of EPaxos to facilitate apples-to-apples comparisons between protocols. Our implementation is a multi-object storage system that uses the protocols as described in this paper with the addition of object identifiers to messages and state. Our code and experiment scripts are available online [29]. We use the existing implementation of MultiPaxos in the framework for our experiments. All of our experiments use the thrifty optimization for EPaxos, MultiPaxos, and Gryff. We use the read proxy optimization for Gryff.

6.2 Experimental Setup

Testbed. We run our experiments on the Emulab testbed [61] using pc3000 nodes. These node types have 1 Dual-Core 3 GHz CPU, 2 GB RAM, and 1 Gbps links to all other nodes. For three replica latency experiments, we emulate replicas in California (CA), Virginia (VA), and Ireland (IR). In five replica latency experiments, we add replicas in Oregon (OR) and Japan (JP). In all experiments, we place the MultiPaxos leader in CA.

We emulate wide-area network latencies using Linux's Traffic Control (tc) to add delays to outgoing packets on all nodes. Table 5 shows the configured round-trip times between nodes in different regions. We choose these numbers because they are the typical round-trip times between the corresponding Amazon EC2 availability regions.

Clients. For all experiments, we use 16 clients co-located with each replica. This number of clients provides enough load on the evaluated protocols to observe the effects of concurrent operations from many clients, but only moderately saturates the system. We avoid full saturation in order to isolate the protocol mechanisms that affect tail latency from hardware and software limitations at various levels in our stack. Clients perform operations in a closed loop.



(a) 2% conflicts. (b) 10% conflicts. (c) 25% conflicts.

Figure 6: Gryff's reads always complete in 1 RTT when $n = 3$. 99th percentile read latency is between 0 ms and 115 ms lower than EPaxos and 134 ms lower than MultiPaxos.

Measurement. Each experiment is run for 180 seconds and we exclude results from the first 15 seconds and last 15 seconds to avoid artifacts from start-up and cool-down. The latency for an individual operation is measured as the time between when a client invokes the operation and when it is notified of the operation's completion.

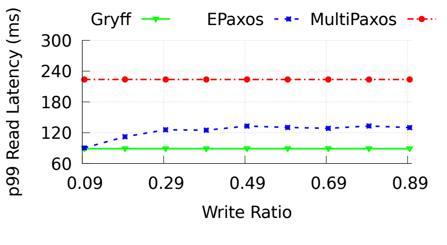
Conflicting Operations. When two operations target the same object in a storage system, we say the operations *conflict*. We use *conflict percentage* as a parameter in our workloads to control the percentage of operations from each client that target the same key. Workloads are highly skewed if and only if their conflict percentage is high.

6.3 Tail Latency

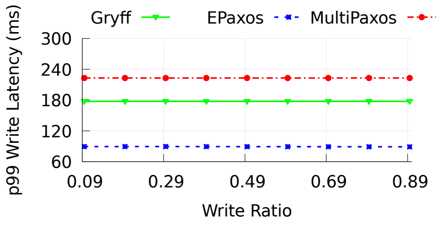
Gryff is designed to reduce the latency cost of linearizability for large scale web applications. Tail latency is of particular importance for these applications because end-user requests for high-level application objects typically fan-out into hundreds of sub-requests to storage services [2, 18]. The object can only be returned to the end-user once all of these sub-requests complete, so the median latency experienced by the end-user is dictated by the tail of the latency distribution for operations to these storage services.

6.3.1 Varying Conflict Percentage

To understand the read tail latency of Gryff and the baselines, we use a variant of the YCSB-B [16] workload that contains 94.5% reads, 4.5% writes, and 1.0% rmws. We examine a read-heavy distribution of operations because most large-scale web applications are read-heavy. For example, more than 99.7% of operations are reads in Google's advertising backend, F1 [17], 99.8% of operations in Facebook's TAO system are reads [10], and 3 out of 5 of YCSB's core workloads contain over 95% reads [16].



(a) Read operations.

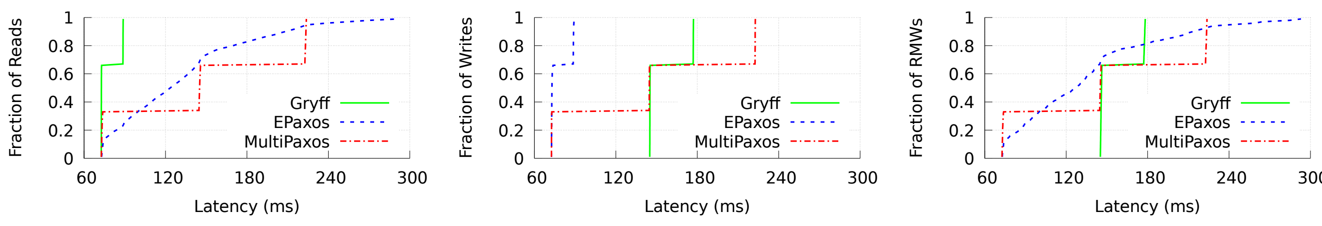


(b) Write operations.

Figure 7: Gryff reduces p99 read latency between 1 ms and 44 ms relative to EPaxos and 134 ms relative to MultiPaxos for varying write percentages. EPaxos' p99 write latency is 89 ms lower than Gryff's p99 write latency regardless of write percentage and conflicts.

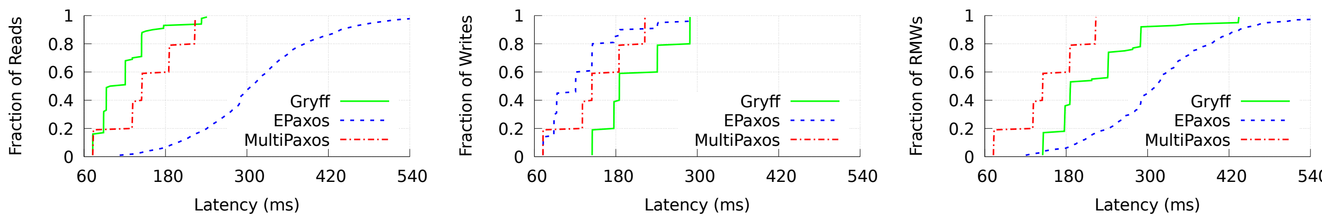
Figure 6a shows the results for three different conflict percentages with $n = 3$. In each sub-figure, a log-scale CDF up to p99.99 is shown below the normal-scale CDF.

1 RTT Reads for Gryff. For $n = 3$ replicas, Gryff always completes reads in 1 RTT due to the read proxy optimization (§5). Figure 6 shows that clients in each region receive responses to their read requests after 1 RTT to the nearest quorum regardless of conflict percentage. Clients in CA are closest to the replicas in CA and VA and vice versa for clients in VA. This results in 66% of the reads completing in the round-trip time between CA and VA (72 ms). Clients in IR



(a) Reads. **(b) Writes.** **(c) RMWs.**

Figure 8: Gryff's writes take 2 RTT, which is always more than EPaxos when $n = 3$. MultiPaxos writes can be faster or slower than Gryff depending on client location and geographic setup.



(a) Reads. **(b) Writes.** **(c) RMWs.**

Figure 9: Gryff trades off worse write latency for better read and rmw latency relative to EPaxos when $n = 5$.

are closest to the replicas in IR and VA, so 33% of the reads complete in the round-trip time between IR and VA (88 ms).

Execution Dependencies Delay EPaxos. EPaxos always commits in 1 RTT for $n = 3$. However, a read cannot complete until a replica executes it and a replica can only execute it after receiving and executing its dependencies. This increases latency when a locally committed read has dependencies on operations that have not yet arrived at the local replica from other replicas. As shown in Figure 6a, these delays do not affect the p99 read latency of EPaxos when there are few conflicts. However, the log-scale CDF shows that a small number of reads are, in fact, delayed.

MultiPaxos has Client-dependent Stable Latency. The MultiPaxos leader can always commit and execute operations in 1 RTT to the nearest quorum. However, clients must also incur a 1 RTT delay to the leader. For clients co-located with the leader (in CA), this delay is negligible, so the latency experienced by these clients with MultiPaxos is less than or equal to the latency experienced with the other protocols. This is demonstrated in the 33rd percentile latencies in Figure 6. For clients not co-located with the leader, the latency is roughly 2 RTT.

Gryff improves 99th percentile read latency between 0 ms and 115 ms relative to EPaxos for low and high conflict percentages and 134 ms relative to MultiPaxos.

6.3.2 Varying Write Percentage

While Gryff's read tail latency is low for read-heavy workloads, we also quantify the tail latency under balanced and write-heavy workloads. To do so, we fix the conflict percentage at 2% and measure the 99th percentile latency of read and write operations for workloads containing 1% rmws and varying ratios of reads and writes. We vary the write percentage from 9.5% to 89.5% and the read percentage from 89.5% to 9.5%. Figure 7 shows the results for $n = 3$ replicas.

Gryff and MultiPaxos Unaffected. The write percentage does not affect Gryff's write latency because its write protocol arbitrarily orders concurrent writes. Similarly, MultiPaxos commits writes through the same path regardless of conflicting operations.

EPaxos Reads Slowdown. With increasing write percentage, the chance that a read obtains a dependency increases even with a fixed conflict percentage (Figure 7a). Unlike reads, writes do not need to be executed before they complete, so they still complete as soon as they are committed. This only takes 1 RTT in EPaxos when $n = 3$. EPaxos dominates Gryff and MultiPaxos for p99 write latency.

Five Replica Varying Write Ratio. We run the same workload with $n = 5$ and show the results in Figure 12 in Appendix A. Gryff can no longer always complete reads in 1 RTT, but due to the low conflict percentage it still achieves a p99 read latency of 1 RTT regardless of write percentage. EPaxos can no longer always commit in 1 RTT. This especially impacts EPaxos' p99 write latency, which becomes approximately the same as Gryff (290 ms).

6.4 Read/Write/RMW Latency

We also quantify the latency distributions of write and rmws in Gryff relative to that of the baselines. For these experiments, we use a variant of the YCSB-A workload with 49.5%

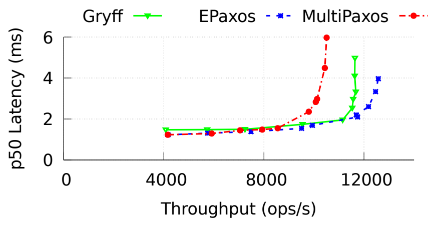


Figure 10: Gryff's throughput at saturation is within 7.5% of EPaxos and is higher than MultiPaxos.

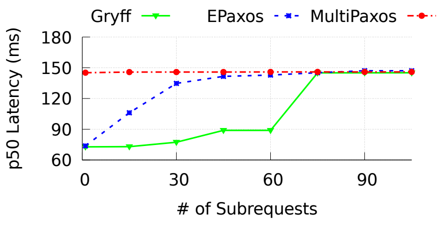


Figure 11: Gryff improves service-level p50 latency when the expected tail-at-scale request contains many reads.

reads, 49.5% writes, and 1.0% rmws with 25% conflicts. The balance between reads and writes allows us to observe the effects that interleavings of operations with different semantics have on the performance of the evaluated protocols. Similarly, the high conflict percentage reveals performance when concurrent operations to the same object interleave.

Figure 8 shows the cumulative distribution functions of the latencies for each operation type for $n = 3$ replicas. Figure 9 shows the same for $n = 5$.

1 RTT Reads for Gryff. For $n > 3$, Gryff often completes reads in 1 RTT, but sometimes takes 2 RTT. Figure 9a demonstrates this behavior as the tail surpasses the 1 RTT latency for any region.

EPaxos Writes are Fast, Reads are Slower. EPaxos dominates Gryff and MultiPaxos for write latency because it always commits in a single round trip for $n = 3$ (Figure 8b) and often commits in a single round trip for $n = 5$ (Figure 9b). As discussed in Section 6.3.1, reads cannot complete until they are executed, so when there are more replicas and more concurrent writes, EPaxos' read latency increases due to the increased likelihood that reads acquire dependencies on updates from other regions.

2 RTT Writes for Gryff. Writes in Gryff takes 2 RTT to complete. Figure 8b demonstrates the gap between EPaxos and Gryff for $n = 3$. When $n > 3$ replicas (Figure 9b), EPaxos still typically completes writes faster than Gryff because it only takes 2 RTT when conflicting concurrent operations arrive at replicas in the intersections of their fast quorums in different orders.

Less Blocking for RMWs in Gryff. Gryff achieves 2 RTT rmws when there are no conflicts and 3 RTT when there are. While Gryff must still block the execution of rmws until all dependencies have been received and executed, Gryff experiences significantly less blocking than EPaxos. This is because EPaxos needs to have dependencies on writes, but Gryff's rmw protocol does not.

EPaxos dominates Gryff for write latency. For $n = 3$, the p50 write latency of Gryff is 72 ms higher and the p99 write latency is 89 ms higher than EPaxos.

6.5 Throughput

We measure median latency at varying levels of load in a local-area cluster. Again, we use the variant of YCSB-A with 49.5% reads, 49.5% writes, and 1.0% rmws with 25% conflicts. Figure 10 shows the results for $n = 3$. We find that Gryff's throughput at saturation is about 11,600 ops/s, within 7.5% of EPaxos. This is also about 1,200 ops/s higher than the maximum throughput of MultiPaxos. Like EPaxos, Gryff does not require a single replica to be involved in the execution of every operation, so it achieves better scalability and load-balancing than leader-based protocols.

Gryff Scales Better. We run the same workload with $n = 5$ and show the results in Figure 13 in Appendix A. Gryff's maximum throughput is higher than EPaxos because EPaxos can no longer always commit on the fast path. Each operation that commits on the slow path on EPaxos requires an additional quorum of messages and replies, which causes the system to more quickly saturate.

6.6 Tail at Scale

Our primary experiments show that Gryff improves read latency relative to our baselines. However, p50 write and p50 rmw latency are lower in EPaxos for $n = 3$. For other parts of the distributions and for MultiPaxos, the latency tradeoff is not comparable. To understand how these tradeoffs with EPaxos and MultiPaxos affect the performance of large-scale web applications whose structure resembles the common structure discussed in Section 6.3, we ran experiments that emulate end-user requests.

We emulate the request pattern of an application preparing a high-level object for an end-user. The object is composed of m sub-requests to the storage system that are drawn from a fixed distribution of reads, writes, and rmws. For example, in order to display a profile page in a social network, dozens of requests to the storage systems that store profile information must be initiated simultaneously [10]. The latency of one of these *tail at scale requests* is the maximum latency of all of its sub-requests. Thus, the median latency of tail at scale requests depends on the tail latency of the sub-requests.

The large-scale web applications whose workloads we emulate are typically read-heavy (§6.3). Moreover, they are often highly skewed. Facebook engineers report that a small set of objects account for a large fraction of total read and write operations in the social graph [2]. This experiment uses a 99%/0.9%/0.1% read/write/rmw workload with 25% conflicts. We vary the number of sub-requests m from 1 to 105 in increments of 15. Figure 11 summarizes the results.

Fast Reads Improve Median End-to-end Latency. Gryff's median latency is lower than that of EPaxos and MultiPaxos when fewer than half of the tail at scale requests are expected to contain a write or rmw operation. Compared to EPaxos' p50 latency, Gryff's is up to 57 ms lower for $n = 3$.

Five Replica Tail-at-scale. We run the same workload with $n = 5$ and show the results in Figure 14 in Appendix A. All protocols follow trends similar to the $n = 3$ case. However, Gryff cannot always complete reads in 1 RTT, so the longer tail of the read latency distribution causes the median latency of these tail at scale requests to increase at a smaller number of sub-requests. Similarly, EPaxos can no longer always commit in 1 RTT, so its tail latency is 2 RTTs plus the delay from blocking for dependencies.

7 Related Work

We review related work in geo-replicated storage systems and combining consensus with shared registers.

EPaxos. EPaxos [47] is the state-of-the-art for linearizable replication in geo-replicated settings. Our evaluation shows that EPaxos dominates Gryff for blind write latency. On the other hand, Gryff dominates EPaxos for read latency and its rmw latency ranges from higher to lower as the contention in the workload increases. This tradeoff is possible because Gryff only uses consensus for operations that require it.

Read Leases. Read leases allow clients to read replicated state from leaseholders by requiring updates to the replicated state be acknowledged by the leaseholders before completing [28, 49]. While this enables reads that need only communicate with a single replica, it sacrifices write availability when a leaseholder fails until the lease expires. Furthermore, to implement read leases safely, clocks at each process must have bounded skew, which is not satisfied by current commodity clocks [25]. Given these difficult availability and safety tradeoffs, we do not consider read leases in the context of Gryff or the baseline systems, but we believe they can be adapted to Gryff's write and rmw protocols.

Other Linearizable Protocols. Paxos [36], VR [50], Fast Paxos [38], Generalized Paxos [37], Mencius [45], Raft [51], Flexible Paxos [33], CAESAR [4], and SD Paxos [62] are consensus protocols that are used to implement linearizable replicated storage systems by ensuring the Agreement property for state machine replication [56]. Other systems, such as Sinfonia [1] and Zookeeper [34], use similarly expensive coordination protocols (2PC and atomic broadcast respectively) to provide strong consistency. CURP [53], Chain Replication [59], and other primary-backup protocols [3] achieve good performance when failures are detectable. Gryff guarantees linearizability in systems with undetectable failures for reads, writes, and rmws and only incurs expensive coordination overhead when needed.

ABD [5] provides linearizable reads and writes with guaranteed termination in asynchronous settings. Subsequent work has established the conditions under which linearizable shared register protocols can provide fast—i.e., complete in 1 RTT—reads [20] or writes [22]. Gryff maintains the performance benefits of these protocols for reads and writes and incorporates rmws for when application developers need stronger synchronization primitives.

Weaker Semantics for Lower Latency. Other geo-replicated systems eschew strong consistency for weaker consistency models that support lower latency operations. PNUTS [15] provides per-timeline sequential consistency, OCCULT [46], COPS [42], and GentleRain [19] provide causal consistency. ABD-Reg [60] provides regularity. Moreover, some systems provide hybrid consistency: Pileus [58], Gemini [39], and ICG [30] allow some operations to be strongly consistent and other operations to be weakly consistent. Gryff provides linearizability to free developers from reasoning about complex consistency models.

Consensus and Shared Registers. Active Quorum Systems (AQS) [7, 8], to our knowledge, was the first attempt to combine consensus with shared registers. We found that AQS allows for non-linearizable executions because its ordering mechanism is unstable for rmws (Appendix C). In contrast, Gryff uses carstamps to stably order rmws with their base updates while allowing for efficient reads and writes with an unstable order. In addition, Gryff is implemented and empirically evaluated.

Cassandra [44] provides reads and writes with tunable consistency and implements a compare-and-swap for applications that occasionally need stronger synchronization. Unlike Gryff, Cassandra's reads and writes are not linearizable by default and its compare-and-swap is not consistent when operating on data also accessed via reads and writes.

8 Conclusion

Gryff unifies consensus and shared registers with carstamps. This reduces latency by avoiding the cost of consensus for the common case of reads and writes. Our evaluation shows that the reduction in latency for individual operations reduces the median service-level latency to $\sim$60% of EPaxos for large-scale web applications.

Acknowledgments We thank our shepherd, Patrick P. C. Lee, and the anonymous reviewers for their helpful comments. We are grateful to Christopher Hodsdon, Theano Stavrinos, Natacha Crooks, Soumya Basu, Haonan Lu, and Khiem Ngo for their extensive feedback. This work was supported by the National Science Foundation under grants number CNS-1824130 and CNS-1932829.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *ACM Symposium on Operating System Principles (SOSP)*, 2007.

[2] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to Adopting Stronger Consistency at Scale. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[3] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *International Conference on Software Engineering*, 1976.

[4] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up Consensus by Chasing Fast Decisions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[5] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[6] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.

[7] A. Bessani. Active Quorum Systems: Specification and Correctness Proof. Technical report, Technical Report DIFCULTR201002, University of Lisbon, 2010.

[8] A. Bessani, P. Sousa, and M. Correia. Active Quorum Systems. In *Workshop on Hot Topics in System Dependability (HotDep)*, 2010.

[9] K. Birman, G. Chockler, and R. van Renesse. Toward a Cloud Computing Research Agenda. *ACM SIGACT News*, 40(2):68–80, 2009.

[10] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference (ATC)*, 2013.

[11] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[13] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure Coding Objects across Global Data Centers. In *USENIX Annual Technical Conference (ATC)*, 2017.

[14] CockroachDB. `https://www.cockroachlabs.com/`, 2020.

[15] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the VLDB Endowment (PVLDB)*, 2008.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, 2010.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Googles Globally-Distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[18] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[19] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[20] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How Fast can a Distributed Atomic Read be? In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

[21] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[22] B. Englert, C. Georgiou, P. M. Musial, N. Nicolaou, and A. A. Shvartsman. On the Efficiency of Atomic Multi-reader, Multi-writer Distributed Memory. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2009.

[23] etcd. `https://etcd.io/`, 2020.

[24] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[25] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. Exploiting a Natural

Network Effect for Scalable, Fine-grained Clock Synchronization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[26] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman. On the Robustness of (Semi) Fast Quorum-Based Implementations of Atomic Shared Memory. In *International Symposium on Distributed Computing (DISC)*, 2008.

[27] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable Consistency in Scatter. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[28] C. G. Gray and D. R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 1989.

[29] Gryff Implementation. `https://www.github.com/matthelb/gryff/`, 2020.

[30] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[31] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[32] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[33] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[34] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, 2010.

[35] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[36] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[37] L. Lamport. Generalized Consensus and Paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[38] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2): 79–103, 2006.

[39] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[40] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[41] R. J. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical report, Technical Report TR-180-88, Princeton University, 1988.

[42] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[43] N. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *International Symposium on Fault Tolerant Computing*, 1997.

[44] P. Malik and A. Lakshman. Cassandra - A Decentralized Structured Storage System. In *ACM SIGOPS Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2009.

[45] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[46] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[47] I. Moraru, D. G. Andersen, and M. Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.

[48] I. Moraru, D. G. Andersen, and M. Kaminsky. A Proof of Correctness for Egalitarian Paxos. Technical report, Technical Report CMU-PDL-13-111, Carnegie Mellon University, 2013.

[49] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[50] B. M. Oki and B. H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[51] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

[52] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAM-

Cloud. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[53] S. J. Park and J. Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[54] Redis. `https://redis.io/`, 2020.

[55] Riak. `https://riak.com/products/riak-kv/`, 2020.

[56] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[57] K. Taranov, G. Alonso, and T. Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2018.

[58] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.

[59] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[60] M. Vukolić. Quorum Systems: with Applications to Storage and Consensus. *Synthesis Lectures on Distributed Computing Theory*, 3(1):1–146, 2012.

[61] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

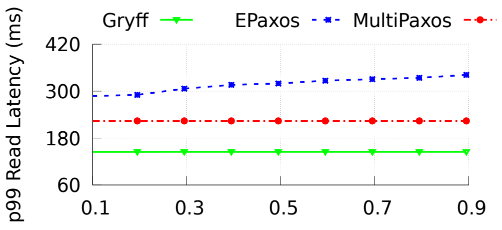
[62] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai. SDPaxos: Building Efficient Semi-Decentralized Geo-Replicated State Machines. In *ACM Symposium on Cloud Computing (SoCC)*, 2018.

A Additional Experiments

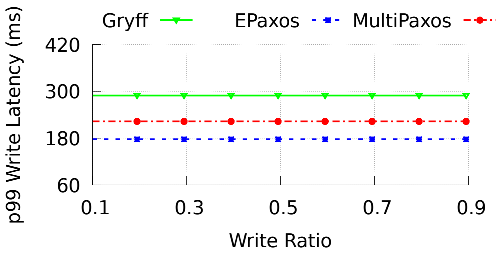
This appendix contains figures for experiments run with $n = 5$ replicas. Discussions of these results are in the main body of the paper in Section 6.

B Proof of Correctness

The proof of correctness for Gryff is presented in five parts. First, we define our model and introduce definitions (§B.1). Second, we describe the remainder of the rmw protocol (§B.2). Third, we prove safety for the base protocol (§B.3).



(a) Read operations.



(b) Write operations.

Figure 12: Gryff has better p99 read latency for $n = 5$ because, even though reads sometimes complete in 2 RTT, enough still complete in 1 RTT that the p99 latency is determined by 2 RTT in a region (CA) where the nearest quorum are relatively close (72 ms per RTT). EPaxos cannot always commit reads or writes in 1 RTT, so its latency increases relative to $n = 3$.

Fourth, we prove liveness for the base protocol (§B.4). Fifth, we argue that the read proxy optimization maintains the desired correctness properties (§B.5).

B.1 Preliminaries

We introduce the system model (§B.1.1) and define a shared object (§B.1.2).

B.1.1 Model

The system is comprised of a set P of *processes* $\{p_1, ..., p_m\}$. A subset $R \subseteq P$ of processes are *replicas* $\{r_1, ..., r_n\}$. Processes communicate with each other over point-to-point message channels. We assume *reliable message delivery*. This abstraction can be implemented on top of unreliable message channels that guarantee eventual delivery via retransmission and deduplication.

Processes may fail according to the *crash failure model*: a failed process ceases executing instructions and its failure is not detectable by other processes. The system is *asynchronous* such that there is no upper bound on the time it takes for a message to be delivered and there is no bound on relative speeds at which processes execute instructions.

Processes are state machines that deterministically transition between states when an *event* occurs. A process interacts with its environment via a set of objects O. The process may

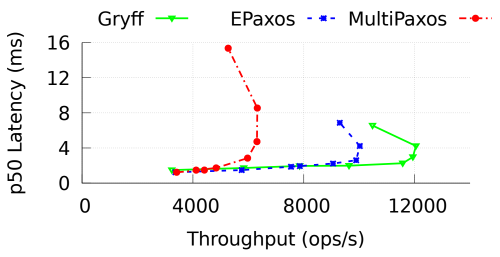


Figure 13: Gryff's throughput at saturation is higher than both EPaxos and MultiPaxos when $n = 5$.

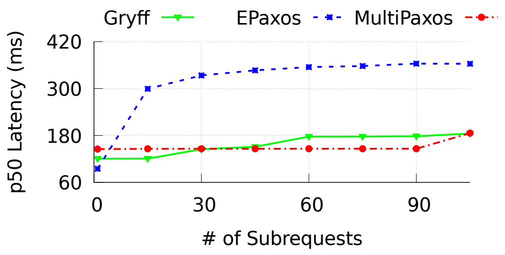


Figure 14: For $n = 5$, the difference in service-level p50 latency is larger because reads in EPaxos suffer from more blocking with more replicas and clients executing operations.

receive an operation op for an object via an invocation event $inv(op)$. The process indicates the result of the operation by generating a response event $resp(op)$. Internal events are the modification of local state at a process, the sending or receipt of a message, and the failure of process. We denote the process associated with an event e by $process(e)$.

An *execution* is an infinite sequence of events generated when the processes run a distributed algorithm. A *partial execution* is a finite prefix of some execution. A process is *correct* in an execution if there are infinite number of events associated with it. Otherwise, the process is *faulty*. Given a set of processes P and an execution e, we denote the set of correct processes in P by $alive(e,P)$, and the set of faulty processes in P by $faulty(e,P)$.

We borrow histories and related definitions from Herlihy and Wing [32]. A *history* h of an execution e is an infinite sequence of operation invocation and response events in the same order as they appear in e. A history may also be defined with respect to a partial execution e'; such a history is a finite sequence. A *subhistory* of a history h is a subsequence of the events of h.

We denote by $ops(h)$ the set of all operations whose invocations appear in h. An invocation is *pending* in a history if no matching response follows the invocation. If h is a history, $complete(h)$ is the maximal subsequence of h consisting only of invocations and matching responses. A history h is *complete* if it contains no pending invocations.

A history h is *sequential* if (1) the first event of h is an invocation and (2) each invocation, except possibly the last, is immediately followed by a matching response and each response is immediately followed by an invocation.

A *process subhistory*, $h|i$, of a history h is the subsequence of all events in h which occurred at p_i. An object subhistory h/o is similarly defined for an object $o \in O$. Two histories h and h' are *equivalent* if $\forall 1 \leq i \leq m. h|i = h'|i$. A history h is *well-formed* if $\forall 1 \leq i \leq m. h|i$ is sequential. We assume all histories are well-formed.

A set S of histories is *prefix-closed* if, whenever h is in S, every prefix of h is also in S. A *single-object* history is one in which all events are associated with the same object. A *sequential specification* for an object $o \in O$ is a prefix-closed set of single-object sequential histories for o. A sequential history h is *legal* if $\forall o \in O. h/o$ belongs to the sequential specification for o.

A history induces an irreflexive partial order on $ops(h)$, denoted $<_h$, as $op_1 <_h op_2$ if and only if $resp(op_1) < inv(op_2)$ in h.

A *quorum system* $\mathcal{Q} \subseteq \mathcal{P}(R)$ over R is a set of subsets of R with the *quorum intersection property*: for all $Q_1, Q_2 \in \mathcal{Q}$, $Q_1 \cap Q_2 \neq \emptyset$. We use *quorum* both to mean a set of replicas in a particular quorum system and the size of such a set.

B.1.2 Shared Objects

A shared object is a data type that supports the following operations:

- READ(): returns the value of the object
- WRITE(v): updates the value of the object to v
- RMW($f(\cdot)$): atomically reads the value v of the object, updates the value to $f(v)$, and returns v

We use $reads(h)$, $writes(h)$, and $rmws(h)$ to denote the set of all operations that are reads, writes, and rmws in $ops(h)$ respectively. We use $updates(h) = writes(h) \cup rmws(h)$ to denote the set of operations which update the state of a shared object in $ops(h)$. We use $observes(h) = reads(h) \cup rmws(h)$ to denote the set of operations which observe the state of a shared object in $ops(h)$.

Definition B.1. *(Shared Object Specification) A sequential object subhistory h/o belongs to the* sequential specification of a shared object *if for each $op \in observes(h/o)$ such that $resp(op) \in h/o$, $resp(op)$ contains the value of the latest preceding operation $u \in updates(h/o)$ or if there is no preceding update, then $resp(op)$ contains the initial value of o.*

B.2 Recovery for RMW Protocol

Algorithms 5 and 6 show the modifications to the basic EPaxos recovery protocol. In addition to the replica state in Figure 4, each replica also maintains *epoch*, the current epoch used in generating ballot numbers, and b, the highest ballot number seen in the current epoch. Each instance in the

Algorithm 5: Recovery coordinator protocol for rmws.

```
1  when replica $r \in R$ suspects replica $c \in R$ failed while committing instance $j$ do
2      $ballot \leftarrow (epoch, (b+1), id_r)$
3      send $Prepare(ballot, id_c, j)$ to all $r \in R$
4      wait to receive $PrepareOK(cmd_r, seq_r, deps_r, base_r, status_r, ballot_r)$ from all $r \in Q \in \mathcal{Q}$
5      $\mathcal{R} \leftarrow \{(cmd_r, seq_r, deps_r, base_r, status_r) \mid \forall r' \in Q : ballot_r \geq ballot_{r'}\}$
6      if $(cmd, seq, deps, base, \textit{committed}) \in \mathcal{R}$ then
7          run Commit Phase for $(cmd, seq, deps, base)$ at $(id_c, j)$
8      else if $(cmd, seq, deps, base, \textit{accepted}) \in \mathcal{R}$ then
9          run Accept Phase for $(cmd, seq, deps, base)$ at $(id_c, j)$
10     else if $\exists S \subseteq \mathcal{R}$ : $(cmd_c, seq_c, deps_c, base_c, status_c) \notin S) \wedge$ $(|S| \geq \lfloor \frac{n}{2} \rfloor) \wedge$ $(\forall reply_1, reply_2 \in S. reply_1 = reply_2 \wedge reply_1.status = \textit{pre-accepted})$ then
11         run Accept Phase for $(cmd_r, seq_r, deps_r, base_r) \in S$ at $(id_c, j)$
12     else if $(cmd, seq, deps, base, \textit{pre-accepted}) \in \mathcal{R}$ then
13         run PreAccept Phase for $cmd$ at $(id_c, j)$, avoid fast path
14     else
15         run PreAccept Phase for $no\text{-}op$ at $(id_c, j)$, avoid fast path
```

cmds array also contains a *ballot* number that is only used during recovery.

Note that the only change Gryff makes is that the *base* attribute is recovered along with the *deps* and *seq* attributes. To support optimized EPaxos, similar changes must be made to the optimized recovery protocol. We refer the reader to the optimized recovery protocol description in the EPaxos technical report [48] and our implementation of Gryff [29] for more details.

B.3 Proof of Linearizability

More Definitions. A *consistency condition* is specified by a particular set of schedules. Linearizability [32] is a strong consistency condition that reduces the complexity of building correct applications.

Definition B.2 (Linearizability)**.** *A complete history h satisfies* linearizability *if there exists a legal total order τ of $ops(h)$ such that $\forall op_1, op_2 \in ops(h). op_1 <_h op_2 \implies op_1 <_\tau op_2$.*

Algorithm 6: Recovery replica protocol for rmws.

```
1  when replica $r \in R$ receives a message $m$ from $x \in R$ do
2      case $m = Prepare(ballot, j, k)$ do
3          if $ballot > cmds[j][k].ballot$ then
4              $cmds[j][k].ballot = ballot$
5              send $PrepareOK(cmds[j][k])$ to $x$
6          else
7              send $NACK$ to $x$
```

Given a particular consistency condition, we are interested in whether a system enforces the condition for all possible partial executions.

Definition B.3. *The system* provides consistency condition *C if, for every partial execution e of the system, the history h of e can be extended to some history h' such that $complete(h')$ is in C.*

Unless otherwise noted, the rest of this section considers a complete history h produced by the distributed algorithm specified in Algorithms 1, 2, 3, and 4 in the main body of the paper and Algorithms 5 and 6 in this appendix.

The *coordinator* of a read or write is the invoking process. For rmws, the coordinator is the replica that notifies the invoking process its rmws has been executed. We assume that each $u \in updates(h)$ writes a unique value.

Definition B.4. *A complete operation $op \in observes(h)$* observes *an update $u \in updates(h)$ if the value returned in $resp(op)$ was written by u.*

Definition B.5. *The* carstamp cs_{op} *assigned to a complete operation $op \in ops(h)$ is:*

- *If $op \in writes(h)$, cs_{op} is the carstamp determined on Line 15 of Algorithm 1.*
- *If $op \in rmws(h)$, cs_{op} is the carstamp determined by Property B.4.*
- *If $op \in reads(h)$, cs_{op} is the carstamp cs_u assigned to the update u that op observes.*

Structure. We abstract the implementation details of the rmw protocol into four sufficient properties. The proofs of the subsequent lemmas and theorem assume that the rmw protocol provides these properties. At the end of this subsection, we prove that Gryff's rmw protocol does exactly this.

Property B.1. *(Freshness) Every complete $rmw \in rmws(h)$ is assigned a carstamp such that $\forall Q \in \mathcal{Q}. cs_{rmw} > \min_{r \in Q} cs_r$ where cs_r is the carstamp at r when rmw is invoked.*

Property B.2. *(Propagation) For every complete $rmw \in rmws(h)$ there exists a $Q \in \mathcal{Q}$ such that $\forall r \in Q. cs_r \geq cs_{rmw}$ where cs_r is the carstamp at r when rmw completes.*

Property B.3. *(Uniqueness) For all complete $rmw_1, rmw_2 \in rmws(h)$, $cs_{rmw_1} \neq cs_{rmw_2}$.*

Property B.4. *(Assignment) Every complete $rmw \in rmws(h)$ is assigned the carstamp $cs_{rmw} = (cs_u.ts, cs_u.id, cs_u.rmwc + 1)$ where u is the update that rmw observes.*

The linearizability proof follows a linear structure. We first prove that the carstamps assigned to each operation respect the real time order of h in Lemmas B.1-B.5. These proofs leverage the quorum intersection property. Then, we prove that a partial order on operations induced by their carstamps respects both the real time order of h and the legality condition for shared objects in Lemmas B.6-B.10. Finally, we connect these lemmas in Theorem B.1 to show that a total order of this partial order satisfies linearizability.

Lemma B.1. *After a replica $r \in R$ executes the* APPLY *function with tuple (v, cs) and before it executes any other instruction, $cs_r \geq cs$ where cs_r is the carstamp at r.*

Proof. By the condition on Line 13 of Algorithm 2. □

Lemma B.2. *$\forall r \in R, cs_r$ monotonically increases where cs_r is the carstamp at r.*

Proof.

1. cs_r is only modified via the APPLY function.

   PROOF: By the fact that, out of all of the replica pseudocode in Algorithms 2, 3, 4, 5, and 6, the APPLY function in Algorithm 2 is the only place that cs_r is assigned a value.

2. Q.E.D.

   PROOF: By Lemma B.1 and 1. □

Lemma B.3. *If an operation $op \in ops(h)$ is complete, then after $resp(op)$ there exists a $Q \in \mathcal{Q}$ such that $\forall r \in Q. cs_r \geq cs_{op}$ where cs_r is the carstamp at r.*

Proof.

1. Let op be an operation in $ops(h)$
2. CASE: $op \in writes(h)$

   2.1. Let $Q \in \mathcal{Q}$ be the quorum from which the coordinator of op receives *Write2Reply* messages.

   PROOF: By the hypothesis that op is complete and the requirement that the coordinator of op waits to receive *Write2Reply* messages from a quorum before completing op (Line 17 of Algorithm 1).

   2.2. Each $r \in Q$ received a *Write2* message for op containing (v, cs_{op}) where v is the value written by op.

   PROOF: By 2.1 and that a replica sends a *Write2Reply* message for op to the coordinator of op only if it receives a *Write2* message for op containing (v, cs_{op}).

   2.3. Each $r \in Q$ applied (v, cs_{op}) before sending a *Write2Reply* message for op.

   PROOF: By 2.1, 2.2, and the requirement that a replica sends a *Write2Reply* message after it applies the tuple it received in a *Write2* message (Line 10 of Algorithm 2).

   2.4. Q.E.D.

   By Lemma B.1, Lemma B.2, and 2.3.

3. CASE: $op \in reads(h)$

   3.1. CASE: op completed after Read Phase 1 (Line 7 of Algorithm 1).

   3.1.1. Let $Q \in \mathcal{Q}$ be the quorum from which the coordinator of op receives *Read1Reply* messages.

   PROOF: By the hypothesis that op is complete and the requirement that the coordinator of op waits to receive *Read1Reply* messages from a quorum before completing op (Line 3 of Algorithm 1).

   3.1.2. When each $r \in Q$ sent their *Read1Reply* message, $cs_r = cs_{op}$ where cs_r is the carstamp at r.

   PROOF: By 3.1.1, Definition B.5, the case 3.1 assumption, and the fast read condition (Line 6 of Algorithm 1).

   3.1.3. Q.E.D.

   PROOF: By Lemma B.2 and 3.1.2.

   3.2. CASE: op completed after Read Phase 2 (Line 10 of Algorithm 1).

   3.2.1. Let $Q \in \mathcal{Q}$ be the quorum from which the coordinator of op receives *Read2Reply* messages.

   PROOF: By the hypothesis that op is complete, the case 3.2 assumption, and the requirement that the coordinator of op waits to receive *Read2Reply* messages from a quorum before completing op in Read Phase 2 (Line 9 of Algorithm 1).

   3.2.2. Each $r \in Q$ received a *Read2* message for op containing (v, cs_{op}) where v is the value written by op.

   PROOF: By 3.2.1 and that a replica sends a *Read2Reply* message for op to the coordinator of op only if it receives a *Read2* message for op containing (v, cs_{op}).

   3.2.3. Each $r \in Q$ applied (v, cs_{op}) before sending a *Read2Reply* message.

   PROOF: By 3.2.1, 3.2.2, and the requirement that a replica sends a *Read2Reply* message after it applies the tuple it received in a *Read2* message (Line 5 of Algorithm 2).

3.2.4. Q.E.D.

By Lemma B.1, Lemma B.2, and 3.2.3.

4. CASE: $op \in rmws(h)$

PROOF: By Property B.2.

5. Q.E.D.

PROOF: By 1, 2, 3, and 4. □

Lemma B.4. *For all operations $op \in ops(h)$ and updates $u \in updates(h)$, $op <_h u \implies cs_{op} < cs_u$.*

Proof.

1. Let $Q_{op} \in \mathcal{Q}$ be a quorum such that $\forall r \in Q_{op}. cs_r \geq cs_{op}$ where cs_r is the carstamp at r when u is invoked.

PROOF: By the hypothesis that *op* completed before *u* was invoked and Lemma B.3.

2. Let u be an update in $updates(h)$.
3. CASE: $u \in writes(h)$

3.1. Let $Q_u \in \mathcal{Q}$ be the quorum from which the coordinator of u receives *Write1Reply* messages and cs_{max} be the largest carstamp contained in these messages.

PROOF: By the hypothesis that u is complete and the requirement that the coordinator of u waits to receive *Write1Reply* messages from a quorum before completing u (Line 13 of Algorithm 1).

3.2. Let $r \in Q_{op} \cap Q_u$ be a replica.

PROOF: By 1, 3.1, and the Quorum Intersection property.

3.3. *op* completed before r received a *Write1* message for u.

PROOF: By the hypothesis that *op* completed before u was invoked and 3.2.

3.4. The *Write1Reply* message that r sent for u contains a carstamp $cs_r \geq cs_{op}$.

PROOF: By 1 and 3.3.

3.5. The coordinator for u assigns u the carstamp $cs_u = (cs_{max}.ts + 1, id, 0)$ where $cs_{max} \geq cs_r$ and id is the id of the coordinator for u.

PROOF: By 3.1 and the assignment of a carstamp to u (Lines 14 and 15 of Algorithm 1).

3.6. Q.E.D.

PROOF: By 3.4, and 3.5.

4. CASE: $u \in rmws(h)$

PROOF: By 1 and Property B.1.

5. Q.E.D.

PROOF: By 2, 3, and 4. □

Lemma B.5. *For all operations $op \in ops(h)$ and reads $\rho \in reads(h)$, $op <_h \rho \implies cs_{op} \leq cs_\rho$.*

Proof.

1. Let u be the update that ρ observes.

PROOF: By the hypothesis that ρ is complete and Definition B.4.

2. CASE: $u = op$

2.1. $cs_\rho = cs_u = cs_{op}$

PROOF: By the assumption of case 2, 1, and Definition B.5.

2.2. Q.E.D.

PROOF: By 2.1.

3. CASE: $u \neq op$

3.1. Let $Q_{op} \in \mathcal{Q}$ be a quorum such that $\forall r \in Q_{op}. cs_r \geq cs_{op}$ where cs_r is the carstamp at r when ρ is invoked.

PROOF: By the hypothesis that *op* completed before ρ was invoked and Lemma B.3.

3.2. Let $Q_\rho \in \mathcal{Q}$ be the quorum from which the coordinator of ρ receives *Read1Reply* messages and cs_{max} be the largest carstamp contained in these messages.

PROOF: By the hypothesis that ρ is complete and the requirement that the coordinator of ρ waits to receive *Read1Reply* messages from a quorum before completing ρ (Line 3 of Algorithm 1).

3.3. Let $r \in Q_{op} \cap Q_\rho$ be a replica.

PROOF: By 3.1, 3.2, and the Quorum Intersection property.

3.4. *op* completed before r received a *Read1* message for ρ.

PROOF: By the hypothesis that *op* completed before u was invoked and 3.3.

3.5. The *Read1Reply* message that r sent for ρ contains a carstamp $cs_r \geq cs_{op}$.

PROOF: By 3.1 and 3.4.

3.6. The coordinator for ρ chooses u to be the update corresponding to cs_{max}.

PROOF: By 3.2 and the selection of an update to observe for ρ (Lines 4 and 5 of Algorithm 1).

3.7. Q.E.D.

PROOF: By 3.5 and 3.6.

4. Q.E.D.

PROOF: By 1, 2, and 3. □

We define the relation $<_\psi$ on $ops(h)$ as follows:

- $\forall op_1, op_2 \in ops(h).\, cs_{op_1} < cs_{op_2} \implies op_1 <_\psi op_2$.
- $\forall \rho \in reads(h)$ such that ρ observes an update $u \in updates(h)$, $u <_\psi r$. $\forall u' \in updates(h)$ such that $u <_\psi u'$, $r <_\psi u'$.
- $\forall \rho_1, \rho_2 \in reads(h)$ such that ρ_1 and ρ_2 observe the same update u, $inv(\rho_1) < inv(\rho_2) \implies \rho_1 <_\psi \rho_2$.
- $\forall op_1, op_2, op_3 \in ops(h).\, op_1 <_\psi op_2 \land op_2 <_\psi op_3 \implies op_1 <_\psi op_3$.

Less formally, $<_\psi$ orders operations by their carstamps and inserts reads in between the updates that the reads observe and subsequent updates.

Lemma B.6. *For all $u_1, u_2 \in updates(h)$, $u_1 <_h u_2 \implies u_1 <_\psi u_2$.*

Proof.

1. $cs_{u_1} < cs_{u_2}$.

PROOF: By the hypothesis that $u_1 <_h u_2$ and Lemma B.4.

2. Q.E.D.

PROOF: By 1 and the definition of $<_\psi$. □

Lemma B.7. *For all $u \in updates(h)$ and $\rho \in reads(h)$, $u <_h \rho \implies u <_\psi \rho$.*

Proof.

1. $cs_u \le cs_\rho$.

PROOF: By the hypothesis that $u <_h \rho$ and Lemma B.5.

2. CASE: $cs_u < cs_\rho$.

PROOF: By the definition of $<_\psi$.

3. CASE: $cs_u = cs_\rho$.

3.1. ρ observes u

PROOF: By the assumption of case 3 and Definition B.4.

3.2. Q.E.D.

PROOF: By 3.1 and the definition of $<_\psi$.

4. Q.E.D.

PROOF: By 1, 2, and 3. □

Lemma B.8. *For all $\rho \in reads(h)$ and $u \in updates(h)$, $\rho <_h u \implies \rho <_\psi u$.*

Proof.

1. $cs_\rho < cs_u$.

PROOF: By the hypothesis that $\rho <_h u$ and Lemma B.4.

2. Q.E.D.

PROOF: By 1 and the definition of $<_\psi$. □

Lemma B.9. *For all $\rho_1, \rho_2 \in reads(h)$, $\rho_1 <_h \rho_2 \implies \rho_1 <_\psi \rho_2$.*

Proof.

1. $cs_{\rho_1} \le cs_{\rho_2}$.

PROOF: By the hypothesis that $\rho_1 <_h \rho_2$ and Lemma B.5.

2. CASE: $cs_{\rho_1} < cs_{\rho_2}$

PROOF: By the definition of $<_\psi$.

3. CASE: $cs_{\rho_1} = cs_{\rho_2}$

3.1. $resp(\rho_1) < inv(\rho_2)$

PROOF: By the hypothesis that $\rho_1 <_h \rho_2$.

3.2. $inv(\rho_1) < inv(\rho_2)$

PROOF: By 3.1.

3.3. Q.E.D.

PROOF: By 3.2 and the definition of $<_\psi$.

4. Q.E.D.

PROOF: By 1, 2, and 3. □

Lemma B.10. *If τ is a topological sort of $<_\psi$, τ is a legal total order of $ops(h)$.*

Proof.

1. Let $op \in observes(h)$ be an operation that observes an update $u \in updates(h)$.

PROOF: By the hypothesis that op is completed.

2. CASE: $op \in reads(h)$.

2.1. There is no u' such that $u <_\psi u' <_\psi op$.

PROOF: By the assumption of case 2 and the definition of $<_\psi$.

2.2. There is no u' such that $u <_\tau u' <_\tau op$.

PROOF: By the hypothesis thatτ is a topological sort of $<_\psi$ and 2.1.

2.3. Q.E.D.

PROOF: By 2.2, the definition of legal, and Definition B.1.

3. CASE: $op \in rmws(h)$.

3.1. $cs_{op} = (cs_u.ts, cs_u.id, cs_u.rmwc + 1)$.

PROOF: By Property B.4.

3.2. SUFFICES ASSUME: $\exists u' \in updates(h)$ with carstamp $cs_{u'}$ such that $u <_\psi u' <_\psi op$.
PROVE: False.

3.2.1. $cs_u < cs_{u'} < cs_{op}$.

PROOF: By assumption 3.2 and the definition of $<_\psi$.

3.2.2. CASE: $cs_u.ts < cs_{u'}.ts$

3.2.2.1. $cs_{op}.ts < cs_{u'}.ts$.

PROOF: By the assumption of case 3.2.2 and 3.1.

3.2.2.2. Q.E.D.

PROOF: By 3.2.2.1 and 3.2.1.

3.2.3. CASE: $cs_u.ts = cs_{u'}.ts$ and $cs_u.id < cs_{u'}.id$.

3.2.3.1. $cs_{op}.ts = cs_{u'}.ts$ and $cs_{op}.id < cs_{u'}.id$.

PROOF: By the assumption of case 3.2.3 and 3.1.

3.2.3.2. Q.E.D.

PROOF: By 3.2.3.1 and 3.2.1.

3.2.4. CASE: $cs_u.ts = cs_{u'}.ts$, $cs_u.id = cs_{u'}.id$, and $cs_u.rmwc < cs_{u'}.rmwc$.

3.2.4.1. $cs_{op}.ts = cs_{u'}.ts$ and $cs_{op}.id = cs_{u'}.id$.

PROOF: By the assumption of case 3.2.4 and 3.1.

3.2.4.2. $cs_{op}.rmwc = cs_u.rmwc + 1 \leq cs_{u'}.rmwc$.

PROOF: By the assumption of case 3.2.4 and 3.1 and that the *rmwc* component of a carstamp is a natural number.

3.2.4.3. CASE: $u' \in writes(h)$

3.2.4.3.1. $cs_{u'}.rmwc = 0$

PROOF: By the assignment of a carstamp to u' (Lines 14 and 15 of Algorithm 1).

3.2.4.3.2. Q.E.D.

PROOF: By 3.2.4.3.1 and 3.2.4.2.

3.2.4.4. CASE: $u' \in rmws(h)$

3.2.4.4.1. $cs_{op}.rmwc \neq cs_{u'}.rmwc$.

PROOF: By 3.2.4.1 and Property B.3.

3.2.4.4.2. $cs_{op}.rmwc < cs_{u'}.rmwc$.

PROOF: By 3.2.4.4.1 and 3.2.4.2.

3.2.4.4.3. Q.E.D.

PROOF: By 3.2.4.1, 3.2.4.4.2, and 3.2.1.

3.2.4.5. Q.E.D.

PROOF: By 3.2.4.3 and 3.2.4.4.

3.3. Q.E.D.

PROOF: By 3.2, the definition of legal, and Definition B.1.

4. Q.E.D.

PROOF: By 1, 2, and 3. □

Theorem B.1. *The system implements a shared object with linearizability.*
Proof. Consider a partial execution e with history h. Let h' be h with a response for each pending operation in $updates(h)$ appended to h. Let $h'' = complete(h')$.
1. Let op_1 and op_2 be operations in $ops(h'')$. We prove that $op_1 <_h op_2 \implies op_1 <_\psi op_2$.
2. CASE: $op_1, op_2 \in updates(h'')$.

PROOF: By Lemma B.6.

3. CASE: $op_1 \in updates(h'')$ and $op_2 \in reads(h'')$.

PROOF: By Lemma B.7.

4. CASE: $op_1 \in reads(h'')$ and $op_2 \in updates(h'')$.

PROOF: By Lemma B.8.

5. CASE: $op_1, op_2 \in reads(h'')$.

PROOF: By Lemma B.9.

6. Let τ be a topological sort of $<_\psi$ on $ops(h'')$.
7. τ is a legal total order on $ops(complete(h'))$.

PROOF: By 6 and Lemma B.10.

8. Q.E.D.

PROOF: By 1, 2, 3, 4, 5, and 7. □

RMW Properties. In order to prove that Gryff's rmw protocol provides the aforementioned properties, we rely on the correctness of EPaxos [48]. Because replicas act as coordinators for a rmw invoked by other processes, the failure of a replica during a rmw before the invoking process learns of the result may cause the invoking process to submit its rmw to another replica. Replicas must be able to recognize duplicates, only execute the rmw once, and store the result until the invoking process generates a response event.

This issue affects all protocols that rely on a subset of processes to coordinate the execution of operations on behalf of other processes. In Gryff, if a process learns that a pending rmw has been executed by at least one replica, it must ensure that a quorum have executed the rmw before completing it. A replica can ensure this by sending *Commit* messages with the appropriate attributes to all replicas. Replicas that receive *Commit* messages for a rmw they have already executed can immediately reply with an *Executed* message. For brevity, we omit the duplicate execution check for a replica receiving a rmw in Algorithm 3 and assume that if a replica has already executed a rmw, it will skip to Line 20 of Algorithm 3.

We assume the use of the majority quorum system $\mathcal{Q}_{maj}$ such that $\forall Q \in \mathcal{Q}_{maj} \cdot |Q| = \lfloor \frac{n}{2} \rfloor + 1$. This assumption implies each quorum is a subset of a fast quorum and equivalent to a slow quorum in canonical EPaxos.

Definition B.6. *A command γ is* committed *at a replica $r \in R$ if the cmds array at r contains an instance with γ as the command and* **committed** *as the status.*

Lemma B.11. *The system provides Property B.1.*
Proof. Let *rmw* be an operation in *rmws*(h) and $Q \in \mathcal{Q}$ be a quorum.

1. *rmw* committed with attributes that are the union of the attributes computed by each $r \in S$ where $S \supseteq Q'$ for some $Q' \in \mathcal{Q}$.

 1.1. *rmw* commits with basic EPaxos or with optimized EPaxos.

 1.2. CASE: *rmw* commits with basic EPaxos.

 PROOF: By Step 1.1 of the proof of Theorem 4 in the EPaxos technical report, which states that *rmw* is committed with the union of attributes from $\lfloor \frac{n}{2} \rfloor + 1$ replicas, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

 1.3. CASE: *rmw* commits with optimized EPaxos.

 There are two sub-cases:

 1.3.1. CASE: *rmw* commits without running the recovery procedure.

 PROOF: By 1.2 and that a fast quorum in optimized EPaxos is larger than a majority quorum because this case reduces to 1.2 with the fast quorum size reduced from $n-1$.

 1.3.2. CASE: *rmw* commits through the optimized recovery procedure.

 1.3.2.1. CASE: *rmw* commits before step 7 of the optimized recovery procedure, or after exiting one of the Else branches in step 7.

 PROOF: By Step 2.1 of Theorem 7 of the EPaxos technical report, which states that *rmw* must have been pre-accepted by a majority of replicas, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

 1.3.2.2. CASE: *rmw* committed after exiting the optimized recovery procedure on the If branch in step 7.

 PROOF: By Step 2.2.2 of Theorem 7 of the EPaxos technical report, which states that *rmw* must have been pre-accepted by a majority of replicas, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

 1.3.2.3. Q.E.D.

 PROOF: By 1.3.2.1 and 1.3.2.2.

 1.3.3. Q.E.D.

 PROOF: By 1.3.1 and 1.3.2.

 1.4. Q.E.D.

 PROOF: By 1.1, 1.2, and 1.3.

2. The *base* attribute of *rmw* is chosen such that $base.cs \geq \max_{r \in S} cs_r \geq \max_{r \in Q'} cs_r$ where cs_r is the carstamp at r when *rmw* is invoked.

 2.1. *rmw* committed after the PreAccept Phase or the Accept Phase. Note that the basic recovery procedure and optimized recovery procedure always exit by running the PreAccept, Accept, or Commit phase. Each of these is reducible to committing after the PreAccept phase or Accept phase.

 2.2. CASE: *rmw* committed after the PreAccept Phase (Line 12 of Algorithm 3).

 2.2.1. When each $r \in S$ sent their *PreAcceptOK* message, $cs_r = base.cs$ where cs_r is the carstamp at r.

 PROOF: By 1, the case 2.2 assumption, and the fast path condition (Line 10 of Algorithm 3).

 2.2.2. Q.E.D.

 PROOF: By Lemma B.2 and 2.2.1.

 2.3. CASE: *rmw* committed after the Accept Phase.

 2.3.1. CASE: The Accept phase is run during normal processing.

 PROOF: By Lemma B.2 and the selection of *base* in the Accept Phase (Line 15 of Algorithm 3).

 2.3.2. CASE: The Accept phase is run during recovery (either basic or optimized).

 PROOF: By the fact that the recovery procedures exit directly to the Accept phase only if *rmw* has previously been pre-accepted by a majority.

 2.4. Q.E.D.

PROOF: By 2.1, 2.2, and 2.3.

3. $base.cs \geq \min_{r \in Q} cs_r$.

PROOF: By 2 and the Quorum Intersection property ($\max_{r \in Q'} cs \geq \min_{r \in Q \cap Q'} cs_r \geq \min_{s \in Q'} cs_r$).

4. $cs_{rmw} > base.cs$.

PROOF: By the generation of the carstamp of *rmw* (Line 18 of Algorithm 4).

5. Q.E.D.

PROOF: By 3 and 4. □

Lemma B.12. *The system provides Property B.2.*

Proof. Let *rmw* be an operation in $rmws(h)$.

1. After *rmw* completes, $\exists Q \in \mathcal{Q}$ such that each $r \in Q$ has executed *rmw*.

PROOF: By the hypothesis that *rmw* is complete and the requirement that the coordinator only completes *rmw* when it has received *Executed* messages from a quorum (Line 21 of Algorithm 3).

2. Each $r \in Q$ applied cs_{rmw}.

PROOF: By 1 and that a replica only sends an *Executed* message for *rmw* if it has applied the carstamp and value of *rmw* (Line 20 of Algorithm 3).

3. Q.E.D.

PROOF: By 2, Lemma B.1, and Lemma B.2. □

Lemma B.13. *The system provides Property B.3.*

Proof. Let rmw_a and rmw_b be operations in $rmws(h)$.

1. Either rmw_a is executed before rmw_b or vice versa.

PROOF: By Theorem 4 and Theorem 7 from the EPaxos technical report, that the logic for determining the *deps* and *seq* attributes of a command remains unchanged from EPaxos, and that the logic for determining the execution order of commands remains unchanged from EPaxos.

2. CASE: rmw_a is executed before rmw_b.

2.1. $cs_{rmw_a} < cs_{rmw_b}$.

2.1.1. For any two interfering commands rmw_a and rmw_b, there is a sequence of zero of more interfering commands that are executed between rmw_a and rmw_b. Let this sequence be $rmw_a = rmw_1, ..., rmw_k = rmw_b$.

PROOF: By Theorem 4 and Theorem 7 from the EPaxos technical report.

2.1.2. Proof by induction on the sequence $rmw_1, ..., rmw_k$.

2.1.2.1. Base case: $k = 2$ (rmw_2 immediately follows rmw_1).

2.1.2.1.1. $prev.cs = cs_{rmw_1}$.

PROOF: By the assumption of the base case 2.1.2.1 and that *prev* is only modified when a rmw is executed (Line 19 of Algorithm 4).

2.1.2.1.2. $cs_{rmw_2} > prev.cs$.

PROOF: By the generation of cs_{rmw_2} to be larger than *prev* at the time that rmw_2 is executed (Lines 15, 16, and 18 of Algorithm 4).

2.1.2.1.3. Q.E.D.

PROOF: By 2.1.2.1.1 and 2.1.2.1.2.

2.1.2.2. ASSUME: $cs_{rmw_1} < cs_{rmw_i}$.
PROVE: $cs_{rmw_1} < cs_{rmw_{i+1}}$.

2.1.2.2.1. $prev.cs = cs_{rmw_i}$.

PROOF: By the assumption that rmw_i was the last rmw to be executed and that *prev* is only modified when a rmw is executed (Line 19 of Algorithm 4).

2.1.2.2.2. $cs_{rmw_{i+1}} > prev.cs$

PROOF: By the generation of $cs_{rmw_{i+1}}$ to be larger than *prev* at the time that rmw_{i+1} is executed (Lines 15, 16, and 18 of Algorithm 4).

2.1.2.2.3. Q.E.D.

PROOF: By 2.1.2.2.1 and 2.1.2.2.2.

2.1.3. Q.E.D.

PROOF: By 2.1.1 and 2.1.2.

2.2. Q.E.D.

PROOF: By 2.1.

3. CASE: rmw_2 is executed before rmw_1.

PROOF: By symmetry with case 2.

4. Q.E.D.

PROOF: By 1, 2, and 3. □

Lemma B.14. *The system provides Property B.4.*

Proof. Let *rmw* be an operation in $rmws(h)$.

1. Let $u \in updates(h)$ be the update that *rmw* observes.

PROOF: By the assumption that *rmw* is complete.

2. Let cs_u be the carstamp chosen on Lines 14 and 16 of Algorithm 4.

PROOF: By 1 and Definition B.4.

3. Q.E.D.

PROOF: By 2, Definition B.5, and the generation of cs_{rmw} (Line 18 of Algorithm 4). □

Lemmas B.11, B.12, B.13, and B.14 imply that Gryff's rmw protocol satisfies the assumptions needed to prove Theorem B.1.

B.4 Proof of Wait-Freedom

More Definitions. Wait-freedom is a strong liveness property that guarantees a correct process can always make progress regardless of concurrent operations invoked by other processes.

Definition B.7. *(Wait-Freedom) A subset $S \subseteq ops(h)$ of operations are* wait-free *in a history h with execution e if $\forall op \in S. process(inv(op)) \in alive(e,P) \implies resp(op) \in h$.*

Unless otherwise noted, the rest of this section considers an execution e with history h produced by the distributed algorithm specified in Algorithms 1, 2, 3, and 4 in the main body of the paper and Algorithms 5 and 6 in this appendix.

We assume that there are $n = 2f + 1$ replicas and that up to f replicas may fail and any number of other processes may fail in e. Thus, we assume the use of the majority quorum system $\mathcal{Q}_{maj}$ such that $\forall Q \in \mathcal{Q}_{maj}. |Q| = f + 1$.

Structure. We first prove that Gryff's reads and writes are wait-free in Theorems B.2 and B.3. To prove wait-freedom for rmws, we discuss why the synchrony assumption must be strengthened from asynchrony to partial synchrony. With this stronger assumption, we restate the liveness property of EPaxos and use this to prove that Gryff's rmws are wait-free in Theorem B.5.

Theorem B.2. *The system provides read wait-freedom.*

Proof. 1. Let op be an operation in $reads(h)$.

2. The coordinator of op is correct.

PROOF: By the definition of a coordinator of a read and by the hypothesis that $process(inv(op)) \in alive(e,P)$.

3. $|alive(e,R)| \geq f + 1$

PROOF: By the assumption that at most f out of $2f + 1$ replicas can fail in any execution.

4. The coordinator sends a *Read1* message for op to every replica $r \in R$.

PROOF: By 2 and Line 2 of Algorithm 1.

5. Each $r \in alive(e,R)$ delivers a *Read1* message for op.

PROOF: By 4, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

6. Each $r \in alive(e,R)$ sends a *Read1Reply* message for op to the coordinator.

PROOF: By 5, the assumption that $r \in alive(e,R)$, and that the message handler for a *Read1* message contains no blocking instructions or conditional branches (Algorithm 2).

7. The coordinator delivers *Read1Reply* messages from a quorum $Q \in \mathcal{Q}$.

PROOF: By 2, 3, 6, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

8. CASE: $\forall r \in Q. cs_r = cs_{max}$

PROOF: By 7, the assumption of the case and that the coordinator generates $resp(op)$ when this assumption holds (Lines 6 and 7 of Algorithm 1).

9. CASE: $\exists r \in Q : cs_r \neq cs_{max}$

9.1. The coordinator sends a *Read2* message for op to every replica $r \in R$.

PROOF: By 2, the assumption of the case, and Line 8 of Algorithm 2.

9.2. Each $r \in alive(e,R)$ delivers a *Read2* message for op.

PROOF: By 9.1, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

9.3. Each $r \in alive(e,R)$ sends a *Read2Reply* message for op to the coordinator.

PROOF: By 9.2, the assumption that $r \in alive(e,R)$, and that the message handler for a *Read2* message contains no blocking instructions or conditional branches on sending a reply (Algorithm 2).

9.4. The coordinator delivers *Read2Reply* messages from a quorum $Q \in \mathcal{Q}$.

PROOF: By 2, 3, 9.3, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

9.5. Q.E.D.

PROOF: By 7, 9.4, and the fact that the coordinator generates a $resp(op)$ after receiving a quorum of *Read2Reply* messages (Line 10 of Algorithm 1).

10. Q.E.D.

PROOF: By 7, 8, and 9. □

Theorem B.3. *The system provides write wait-freedom.*

Proof. 1. Let *op* be an operation in $writes(h)$.

2. The coordinator of *op* is correct.

PROOF: By the definition of a coordinator of a write and by the hypothesis that $process(inv(op)) \in alive(e,P)$.

3. $|alive(e,R)| \geq f+1$

PROOF: By the assumption that at most f out of $2f+1$ replicas can fail in any execution.

4. The coordinator sends a *Write1* message for *op* to every replica $r \in R$.

PROOF: By 2 and Line 12 of Algorithm 2.

5. Each $r \in alive(e,R)$ delivers a *Write1* message for *op*.

PROOF: By 4, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

6. Each $r \in alive(e,R)$ sends a *Write1Reply* message for *op* to the coordinator.

PROOF: By 5, the assumption that $r \in alive(e,R)$, and that the message handler for a *Write1* message contains no blocking instructions or conditional branches (Algorithm 2).

7. The coordinator delivers *Write1Reply* messages from a quorum $Q \in \mathcal{Q}$.

PROOF: By 2, 3, 6, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

8. The coordinator sends a *Write2* message for *op* to every replica $r \in R$.

PROOF: By 2, 7, and Line 16 of Algorithm 2.

9. Each $r \in alive(e,R)$ delivers a *Write2* message for *op*.

PROOF: By 8, the assumption that $r \in alive(e,R)$, and the assumption that the network guarantees eventual reliable message delivery.

10. Each $r \in alive(e,R)$ sends a *Write2Reply* message for *op* to the coordinator.

PROOF: By 9, the assumption that $r \in alive(e,R)$, and that the message handler for a *Write2* message contains no blocking instructions or conditional branches on sending a reply (Algorithm 2).

11. The coordinator delivers *Write2Reply* messages from a quorum $Q \in \mathcal{Q}$.

PROOF: By 2, 3, 10, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

12. Q.E.D.

PROOF: By 11 and the fact that the coordinator generates a $resp(op)$ after receiving a quorum of *Write2Reply* messages (Algorithm 1). □

Note that Theorems B.2 and B.3 rely on our weak network assumption that messages are eventually delivered and do not require any stronger assumptions about the synchrony of the system. Eventual message delivery only precludes infinitely long partitions in the network, which is unlikely to occur in any practical system.

RMW Wait-Freedom. The FLP impossibility result implies that no consensus protocol can provide both safety and liveness in asynchronous systems where processes can fail [24]. Because rmw can solve consensus [31], this also implies that no rmw protocol can provide both.

The rest of this section shows that Gryff's rmw protocol provides wait-freedom if we relax the system model from asynchrony to partial synchrony [21]. In the partial synchrony model, there are two bounds Δ and Φ such that after some unknown point in time during an execution of the system, all messages are delivered within Δ time of when they are sent and all correct processes take at most Φ time between the execution of instructions.

As in the proof of linearizability, we rely on the correctness of EPaxos in the partial synchrony model [48].

Theorem B.4. *EPaxos guarantees with high probability that every proposed command will eventually be committed by every $r \in alive(e,R)$ as long as messages eventually reach their destination before their recipient times out.*

Lemma B.15. *With high probability, every $r \in alive(e,R)$ executes every rmw that commits.*

Proof. Let r be a correct replica, *rmw* be an operation in $rmws(h)$, and D be the transitive closure of the set of dependencies for *rmw* determined by the commit protocol.

1. With high probability, every $rmw' \in D$ eventually commits at r.

PROOF: By Theorem B.4.

2. With high probability, every $rmw' \in D$ is executed at r. Proof by generalized induction on D.

2.1. Base case: $rmw_0 \in D$ is the first rmw committed in e.

PROOF: By the assumption that r is correct, the assumption of the base case 2.1, and that the EPaxos execution algorithm contains no blocking instructions for commands with no dependencies.

2.2. ASSUME: For any $rmw'' \in D$ such that rmw'' is before rmw' in the EPaxos execution order, rmw'' is executed at r.
PROVE: rmw' is executed at r

PROOF: By the assumption that r is correct, the induction hypothesis 2.2, and that the EPaxos execution algorithm only blocks the execution of a command until all of its dependencies have executed.

2.3. Q.E.D.

By 1, 2.1, and 2.2.

3. With high probability, after all $rmw' \in D$ have executed, rmw will be executed.

3.1. CASE: rmw is in its own strongly connected component in the dependency graph.

PROOF: By the execution order specified by the EPaxos execution algorithm, which requires every dependency of a command to be executed before the command is executed.

3.2. CASE: rmw is in a cycle in the dependency graph.

PROOF: By the execution order specified by the EPaxos execution algorithm, which requires that cycles be broken in order of *seq*, and the fact that rmw may be executed before some of its dependencies within the same cycle.

3.3. Q.E.D.

PROOF: By 3.1 and 3.2.

4. Q.E.D.

PROOF: By 2 and 3. □

Theorem B.5. *If there is a point in time after which the system is synchronous with bounds Δ and Φ, the system provides rmw wait-freedom with high probability.*

Proof. Let op be an operation in $rmws(h)$.
1. $|alive(e,R)| \geq f+1$

PROOF: By the assumption that at most f out of $2f+1$ replicas can fail in any execution.

2. With high probability, every $r \in alive(e,R)$ commits an instance containing op.

PROOF: By the hypothesis that there is a finite time after which all messages are delivered within Δ time of when they are sent and Theorem B.4.

3. With high probability, every $r \in alive(e,R)$ executes op.

PROOF: By 2 and Lemma B.15.

4. With high probability, every $r \in alive(e,R)$ sends an *Executed* message for op to the coordinator.

By 3 and that there are no blocking instructions or conditional branches on sending an *Executed* message in the EXECUTE function.

5. With high probability, the coordinator delivers an *Executed* message for op from a quorum $Q \in \mathcal{Q}$.

PROOF: By 1, 4, the assumption that the network guarantees eventual reliable message delivery, and the assumption that the majority quorum system $\mathcal{Q}_{maj}$ is used.

6. Q.E.D.

PROOF: By 5 and the fact that the coordinator generates a $resp(op)$ after receiving a quorum of *Executed* messages. □

B.5 Read Proxy Correctness

Algorithm 7: The modified read coordinator protocol and *Read1* message handler for using the read proxy optimization.

```
1  procedure Coordinator::READ($(v, cs)$) at $p \in P$
2      send $Read1(v, cs)$ to all $r \in R$
3      wait to receive $Read1Reply(v_r, cs_r)$ from all
         $r \in Q \in \mathcal{Q}$
4      for $r \in Q$ do
5          APPLY($(v_r, cs_r)$)
6      $cs_{max} \leftarrow \max_{r \in Q} cs_r$
7      $v \leftarrow v_r : cs_r = cs_{max}$
8      if $\forall r \in Q : cs_r = cs_{max}$ then
9          return $v$
10     send $Read2(v, cs_{max})$ to all $r \in R$
11     wait to receive $Read2Reply$ from all $r \in Q' \in \mathcal{Q}$
12     return $v$
13 when replica $r \in R$ receives a message $m$ from $p \in P$ do
14     case $m = Read1(v', cs')$ do
15         APPLY($(v', cs')$)
16         send $Read1Reply(v, cs)$ to $p$
```

The pseudocode for the read proxy optimization described in Section 5 is in Algorithm 7. We briefly argue that the optimization does not change the correctness proofs.

The optimization changes the definition of the coordinator of a read from the invoking process to the replica that notifies the invoking process of the result of the read. Neither the definition change nor the added logic for the optimization affect the proof of linearizability because the value that a read observes is still chosen to be the one associated with the maximum carstamp on a quorum. Reads can be executed multiple times without affecting the state of the shared object, so it is safe for a client to timeout after a finite time t and forward its read to another replica if it suspects the initial coordinator failed.

The proof of wait-freedom for reads remains the same, but needs a small clarification in the proof of Step 2. Since at most f replicas can fail, a client will eventually forward its read to a correct replica that will complete the read coordinator protocol. This will happen after at most $f \cdot t$ time, which is finite.

C Non-Linearizable AQS Execution

AQS [8] attempts to exploit the same observation that Gryff does about the relationship between shared register and consensus protocols to improve performance under the Byzantine failure model. In Figure 15, we demonstrate an explicit execution of AQS that exhibits non-linearizable behavior.

Here, process p_1 first issues and completes w_1 with $ts = (1,1)$ that is seen by all replicas (Figure 15.1). After this write has completed, process p_2 begins w_2 and sees w_1 with $ts = (1,1)$, so it chooses $ts = (2,2)$ for w_2 (Figure 15.2). This write then pauses, and process p_3 issues rmw_3 to primary s_4. The primary gathers state from all replicas and picks $base\_state = \langle w_1, ts = (1,1)\rangle$ (Figure 15.3). The primary then generates an updated state v_l based on w_1 and sends PRE-PREPARE messages to all replicas. These messages are accepted by all replicas because w_1 is the most recent state they have observed (Figure 15.4). All replicas then broadcast PREPARE messages to all other replicas, and the messages are received and accepted. All replicas then broadcast COMMIT messages (Figure 15.5) and rmw_3 pauses. Process p_2 now finishes w_2 by sending out a second round of messages with $ts = (2,2)$, and all replicas accept and apply this write (Figure 15.6). Shortly after, replicas receive COMMIT messages from all other replicas for rmw_3, forming a commit certificate. All replicas generate $ts_l = succ(ts = (1,1), s_4) = (2,4)$ and apply rmw_3 (Figure 15.7). Process p_4 now issues a read ρ_4, and the read completes in one round, returning $ts = (2,4)$ from rmw_3 (Figure 15.8).

There is no legal total order for this execution because rmw_3 must follow w_1 with no writes in between because rmw_3 picks $base\_state = \langle w_1, ts = (1,1)\rangle$. Thus, rmw_3 must be ordered before w_2. We also must have ρ_4 ordered after both rmw_3 and w_2 because it begins in real time after both operations have finished. The read ρ_4 sees rmw_3, so rmw_3 must be ordered after w_2. Thus, there is no legal total order of operations and linearizability is not satisfied.

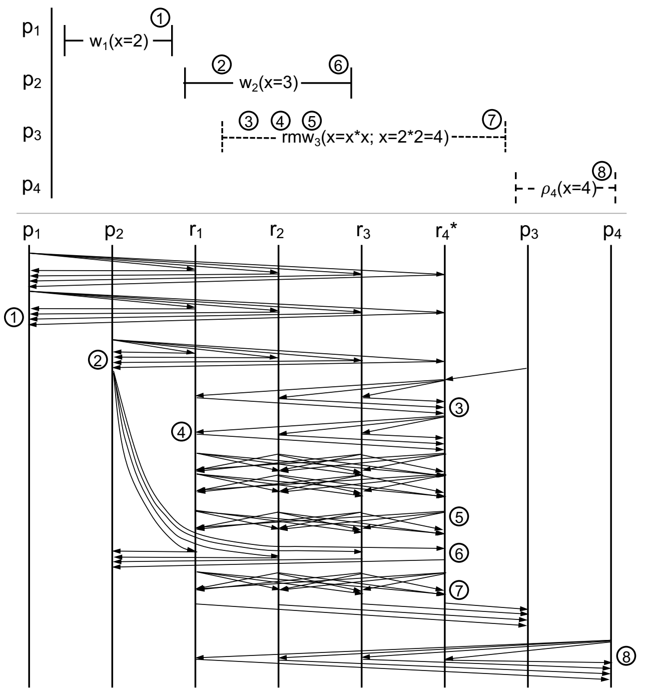


Figure 15: Labeled numbers represent the following events: 1. p_1 issues and completes w_1 with $ts = (1,1)$. 2. p_2 issues w_2 and gets back $ts = (1,1)$; the process then picks $ts = (2,2)$ for w_2. 3. The primary s_4 picks $base\_state = \langle w_1, ts = (1,1)\rangle$. 4. All replicas accept PRE-PREPARE messages because w_1 is the most recent state observed. 5. All replicas broadcast COMMIT messages to all other replicas. 6. All replicas apply w_2 because $ts = (2,2) > ts = (1,1)$. 7. All replicas apply rmw_3 because $ts = (2,4) > ts = (2,2)$. 8. p_4 issues and completes ρ_4 in 1 round, returning rmw_3 with $ts = (2,4)$.

CableMon: Improving the Reliability of Cable Broadband Networks via Proactive Network Maintenance

Jiyao Hu[*]
Duke University

Zhenyu Zhou[*]
Duke University

Xiaowei Yang
Duke University

Jacob Malone
CableLabs

Jonathan W Williams
The University of North Carolina at Chapel Hill

Abstract

Cable broadband networks are one of the few "last-mile" broadband technologies widely available in the U.S. Unfortunately, they have poor reliability after decades of deployment. Cable industry proposed a framework called Proactive Network Maintenance (PNM) to diagnose the cable networks. However, there is little public knowledge or systematic study on how to use these data to detect and localize cable network problems. Existing tools in the public domain have prohibitive high false-positive rates.

In this paper, we propose CableMon, the first public-domain system that applies machine learning techniques to PNM data to improve the reliability of cable broadband networks. CableMon uses statistical models to generate features from time series data and uses customer trouble tickets as hints to infer abnormal thresholds for these generated features. We use eight-month of PNM data and customer trouble tickets from an ISP to evaluate CableMon's performance. Our results show that 81.9% of the abnormal events detected by CableMon overlap with at least one customer trouble ticket. This ticket prediction accuracy is four times higher than that of the existing public-domain tools used by ISPs. The tickets predicted by CableMon constitute 23.0% of the total network-related trouble tickets, suggesting that if an ISP deploys CableMon and proactively repairs the faults detected by CableMon, it can preempt those customer calls. Our current results, while still not mature, can already tangibly reduce an ISP's operational expenses and improve customers' quality of experience. We expect future work can further improve these results.

1 Introduction

Broadband access networks play a crucial role in modern life. They help narrow digital divide, enable e-commerce, and provide opportunities for remote work, study, and entertainment. In the US, cable networks are one of the few available infrastructures that can provide broadband Internet access to US homes. In many rural areas, they are often the only broadband choice. According to a study in 2016 [6], cable broadband is available to 93% of US homes, far more than the two alternative choices: high bitrate digital subscriber line (VDLS) (43%) and Fiber-to-the-Premises (FTTP) (29%).

However, cable networks are prone to reliability problems, partly due to their Hybrid Fiber-Coaxial (HFC) architecture. This architecture uses both optical fibers and coaxial cables to deliver a mixed bundle of traffic, including video, voice, and Internet data. Unlike fiber optics, coaxial cables are vulnerable to radio frequency (RF) interference. Many parts of the cable networks are now decades old [1]. Aging can lead to problems such as cable shielding erosion, loose connectors, and broken amplifiers. All those problems can manifest themselves as poor application-layer performance, e.g., slow web responses or low-quality video streaming. Much measurement study has shown that broadband networks have poor reliability [3,12,17,22,23,28]. A recent one [3] shows that the average availability of broadband Internet access is at most two nines (99%), much less than the minimum FCC's requirement (four nines 99.99%) for the public switched telephone network (PSTN) [20]. Admittedly, if ISPs replace the last-mile coaxial cables with fiber optics, many of these problems may disappear. However, due to the prohibitive cost of FTTP, cable broadband networks are likely to remain as one of the few broadband choices in rural America for the next decade or two. Therefore, it is critically important that the cable Internet services remain robust during emergencies, especially as more and more subscribers migrate their landline phones to VoIP phones.

The cable industry has long recognized this problem and developed a platform called Proactive Network Maintenance (PNM) to improve the reliability of their networks [7]. PNM enables a cable ISP to collect a set of performance metrics from each customer's cable modem. We refer to this set of data as PNM data. One example of a PNM metric is a cable

[*] Jiyao Hu and Zhenyu Zhou, placed in alphabetic order, are the lead student authors and contributed equally to this work.

channel's signal-to-noise ratio. PNM aims to enable an ISP to proactively detect and fix network faults before they impact services and customers.

Although PNM has been incorporated into DOCSIS since 2005 [7], how to use PNM data to improve network reliability remains an open challenge. The best current practice recommended by CableLabs[1] [7] and the tools used by some ISPs [22] use a set of manually configured thresholds to flag a faulty network condition. The feedback from deploying ISPs is that these thresholds are often too conservative, leading to high false positives.

This work aims to improve the reliability of cable broadband networks. We speculate that the challenge of using PNM data is due to the lack of expert knowledge or ground truth on what PNM values warrant a proactive network repair. In an RF system like a cable network, network conditions may degrade gradually, making it challenging to define a static threshold that separates what is faulty from what is not. We develop a system called CableMon, which uses machine learning techniques to infer network faults that demand immediate repair. CableMon couples PNM data with customer trouble tickets to identify the ranges of PNM values that are likely to lead to a customer's trouble call. Our hypothesis is that if a network fault impacts an ISP's customers, then some customer is likely to call to report the problem. Therefore, we can use customer trouble tickets as hints to learn what network conditions are likely to lead to customer trouble calls. It is desirable for an ISP to prioritize its effort to repair those problems, because if they persist, they are likely to reduce customer satisfaction and increase the cost for customer support.

A key technical challenge we face is that both customer tickets data and PNM data contain much noise. Customer tickets are not reliable indicators of network faults. A customer may or may not call when there is an underlying network problem, and vice versa. PNM data, by its nature, describe cable channels' conditions as well as environmental noises. Therefore, if we use customer tickets to label PNM data as normal or faulty, and apply an off-the-shelf machine learning technique to detect network faults, we may not have good detection results. In addition, manual labeling is not practical in this context, because there lacks expert knowledge and the dataset is too large.

In CableMon's design, we use three techniques to address the above challenges (§ 4). First, to reduce noise in customer tickets, we filter customer tickets according to the ticket descriptions and only choose the tickets that suggest network problems as hints. Second, to reduce noise in PNM data, we treat a modem's PNM data as time series data and use its time series features (e.g., expected moving average or variance) for fault detection. Third, we develop a customized classification method that is robust to both noise in tickets and noise in PNM data.

[1]CableLabs is a research and development lab founded by American Cable operators in 1988 and led the development of DOCSIS and PNM.

With the support of CableLabs, we have obtained eight months of anonymized PNM data and the corresponding customer trouble tickets from a mid-size U.S. ISP. We use five months of data to train CableMon, and use the next three months data following the training set as the test set to evaluate how well CableMon detects network faults. CableMon takes the PNM data in our test set as input and detects when a network fault occurs and when it ends. Due to the lack of ground truth, we evaluate CableMon's performance using customer trouble tickets in the test set. When CableMon detects a network anomaly, if a customer who experiences the anomaly reports a ticket, we consider the detection a success. We compare CableMon with a tool currently used by our collaborating ISP, which we refer to as AnonISP, and with a tool developed by Comcast [22]. Our results show that 81.9% of the anomalies detected by CableMon lead to a customer trouble ticket. In contrast, only 10.0% of the anomalies detected by AnonISP's tool lead to a trouble ticket; and 23.5% of the anomalies detected by Comcast's tool lead to a customer ticket. In addition, CableMon predicts 23.0% of all network-related tickets, while AnonISP's tool predicts 25.3% and Comcast's tool predicts less than 3%. The trouble tickets predicted by CableMon on average last 32.5 hours (or 53.3%) longer than those predicted by other tools, suggesting that those tickets are more likely to require repair efforts. The median time from the beginning of a fault detected by CableMon to the reporting time of a ticket is 164.1 hours (or 29.3%) shorter than that of a fault detected by other tools, suggesting that the faults detected by CableMon require more immediate repair.

To the best of our knowledge, this work is the first large-scale public study that couples PNM data with customer trouble tickets to improve the reliability of cable networks. Our main contribution is CableMon, a system that detects network faults more reliably than the existing public-domain work. It serves as a starting point to unleash the full potential of PNM data. We are working with an ISP and the CableLabs to deploy CableMon in practice and we expect the feedback from practice can further improve the performance of CableMon. One general lesson we learn is that one can use customer trouble tickets as hints to learn what values of network performance metrics indicate customer-impacting problems, despite the presence of noise in both the ticket data and the network performance data. We believe this lesson is applicable to proactive network maintenance in other types of networks, including cellular networks, WIFI access networks, and data-center networks.

2 Background and Datasets

In this section, we briefly introduce the cable Internet architecture and describe the datasets we use in this work.

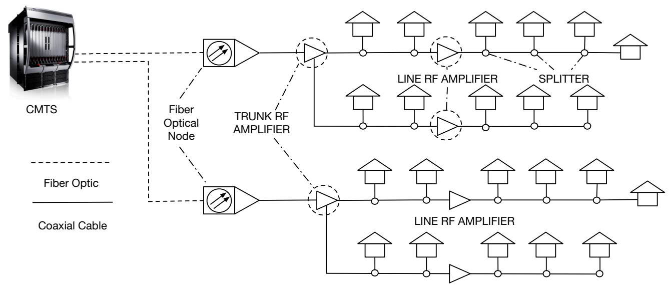


Figure 1: **An overview of the Hybrid Fiber Coaxial (HFC) architecture.**

2.1 Cable Network Architecture

Figure 1 shows a high-level overview of a cable broadband network. A cable broadband network is an access network. It provides the "last-mile" Internet connectivity to end users. A customer connects to the cable network via a cable modem residing in her home. The cable access network terminates at a device called a Cable Modem Termination System (CMTS), which is a router with one port connecting to the Internet and many other ports connecting to customers' cable modems.

At the IP level, there is only one hop between a customer's cable modem/home router and the CMTS. Underlying this single IP hop, there is a complicated link-level structure that consists of different types of physical links and devices. The "last-mile" links that connect to the customer premises are often made of copper coaxial cables. These cables terminate at a device called a fiber node (FN). A fiber node connects to the CMTS via optical fibers. It converts the incoming optically modulated signals into electrical signals and sends the signals towards the customers' homes, and vice versa. Due to signal attenuation, cable networks deploy radio frequency (RF) amplifiers between a fiber node and a residential home. Along the way to a customer's home, new branches may split from the main cable by the line splitters. All these devices could introduce signal distortion and noise.

Historically, cable TV networks divide radio frequency into multiple channels, each of 6MHz width. Cable broadband networks use a subset of these channels as data carriers. A cable ISP typically uses three or four of these channels at the lower end of the spectrum to carry data from a user's cable modem to CMTS. We refer to this direction as the *upstream* direction. An ISP may use sixteen or more of the channels at the higher end of the spectrum to carry data from a CMTS to a modem. We refer to this direction as the *downstream* direction.

2.2 Datasets

We have obtained two types of anonymized modem-level data from a U.S. cable ISP for this study. They include (1) PNM data and (2) customer trouble ticket data. We have a total of eight months of data dating from 01/06/2019 to 08/31/2019. Next, we describe each dataset in turn. [2]

PNM data: The PNM data we obtained were collected by a common standard built into DOCSIS. A CMTS can query a DOCSIS-compliant cable modem (CM) to obtain certain performance data. DOCSIS standardizes how a CM or CMTS stores these performance data in a local Management Information Base (MIB) [7]. A remote process can use the Simple Network Management Protocol (SNMP) to query the MIBs of each CM or a CMTS to obtain performance data [36].

Currently, we only have PNM data from the upstream channels. DOCSIS 3.0 gives a cable operator the ability to collect the full spectrum view of a cable modem's RF channels. It is our future work to investigate whether this type of data may further improve our detection accuracy.

A record in the PNM data we obtain has the following fields:

- *Timestamp*: The time when a PNM query is sent.
- *Anonymized MAC*: The hashed MAC address of the queried CM.
- *Anonymized Account Number*: The hashed user account number. This field is used to link a customer ticket with the corresponding PNM data from the customer's CM.
- *Channel Frequency*: This field identifies which upstream channel this record is about.
- *SNR*: The upstream signal-to-noise ratio of this channel.
- *Tx Power*: A CM's signal transmission power.
- *Rx Power*: The received signal power at the CMTS.
- *Unerrored*: The number of unerrored codewords received at the CMTS.
- *Corrected*: The number of errored but corrected codewords received at the CMTS.
- *Uncorrectable*: The number of errored but uncorrected codewords.
- *T3 Timeouts*: The number of DOCSIS T3 timeouts [5] the CM has experienced since its last reboot. A DOCSIS T3 timeout occurs when there is an error in a CM's ranging process, which we will soon explain.
- *T4 Timeouts*: The number of DOCSIS T4 timeouts [5] the CM has experienced since its last reboot. Similarly, a T4 timeout occurs when there is a ranging error.
- *Pre-Equalization Coefficients*: The set of parameters a CM uses to compute how to compensate for channel distortions during a ranging process.

A CM uses a process called *ranging* to compute a set of parameters called *pre-equalization coefficients* for mitigating channel distortions. When RF signals travel along a coaxial cable, they may be distorted as different frequencies attenuate at different speeds and noise may be added to the channel. To mitigate the channel distortions, a CM adds correction signals to the data signals it transmits. Ideally, the correction signals will cancel out the distortions when the signals arrive at the CMTS. A CM and a CMTS exchange messages periodically

[2]We note that we have discussed this work with our institute's IRB. And they consider it does not involve human subjects.

to compute the correction signals. This process is called ranging. And the set of parameters used to compute the correction signals are called pre-equalization coefficients.

The PNM data we obtain are collected every four hours from several of an ISP's regional markets. There are around 60K unique account numbers in our datasets.

Customer Ticket Data: We have also obtained the records of customer trouble tickets from the same ISP. The relevant fields in each record include the hashed customer's account number, the ticket creation time, the ticket close time (if it was closed), a brief description of the actions taken to resolve the ticket, and a possible diagnosis.

3 Overview

In this section, we motivate the design of CableMon by describing the limitations of existing work. We then describe the design rationale of CableMon, its design goals, and the design challenges we face.

3.1 Limitations of Existing Work

The existing PNM work in the public domain [7, 17, 22] use a set of PNM metrics and manually-set thresholds to detect network faults. If the value of a metric is below or above a threshold, it indicates a fault. This approach has several limitations. First, it is challenging to set the right thresholds. If the thresholds were set too conservatively, they might flag too many faults for an ISP to fix. In contrast, if they were set too aggressively, an ISP might miss the opportunities for proactive maintenance. There lacks a systematic study on how to set the threshold values to achieve the best tradeoff. Second, the existing work mostly uses the instantaneous values of PNM data for fault detection. However, due to the inherent noise in PNM data, using the instantaneous values may lead to instability in detection results. In addition, it may fail to detect faults that can only be captured by abnormalities in a PNM metric's statistical values, e.g., variations.

For ease of explanation, we use one threshold value recommended in the CableLabs' PNM best practice document [7] to illustrate the limitations. CableLabs' recommendation uses a variable called Main Tap Ratio (MTR) computed from a modem's pre-equalization coefficients. It specifies that when the MTR value of a modem is below a threshold (<18dB), there is a fault in the network that needs immediate repair.

We sample the MTR values in one of the ISP's markets. There are more than 60K modems in this market. We choose five random days' records during an eight-month period in 2019 and measure the MTR values of all modems during the sampled days. Table 1 shows the percentage of modems that have a channel whose MTR value is below the recommended threshold. If an ISP used the recommended MTR threshold, at any sampled day, there would be more than 24% of cable modems that require immediate repair. We also measure the MTR values among all PNM records during this eight-month period. In more than 26% of the records, a modem's MTR value is below 18dB.

3.2 Design Goals

CableMon aims to enable an ISP to detect network problems that demand immediate repair. Specifically, it aims to accurately detect the set of network conditions that adversely impact customer experience. We refer to these network conditions as network anomalies or faults in this work. Its design goals include the following:

- High ticket prediction accuracy, and moderate ticket coverage. Ideally, we would like to use precision (the set of true positives detected over all detected positives) and recall (the set of true positives detected over all true positives) to measure the performance of CableMon. However, because we do not know ground truth, we instead use customer tickets as indications of true positives. We define *ticket prediction accuracy* as the ratio between the number of anomalies detected by CableMon that lead to one or more customer tickets and the number of total anomalies CableMon detects. Similarly, we define *ticket coverage* as the ratio between the number of tickets CableMon predicts and the total number of network-related customer tickets. It is desirable that CableMon has high ticket prediction accuracy because an ISP is often limited by the number of technicians it has to repair network faults, avoiding false alarms is practically more important than repairing all faults proactively. What we learned from AnonISP is that even a 10% reduction in customer calls can reduce their operational costs significantly. Therefore, as a starting point, we aim for a high ticket prediction accuracy and a moderate ticket coverage.

- No manual labeling. One approach to detect network anomalies is to train a supervised learning classifier on labeled data. The labels tell what PNM metrics indicate network anomalies and what do not. However, we do not have such labeled data. And due to lack of ground truth and the large size of the data, manual labeling is also practically challenging. Therefore, we aim to design CableMon without requiring manual labeling.

- No extensive parameter tuning. We aim to release CableMon as an off-the-shelf-tool at cable ISPs. Therefore, we require that CableMon's fault detection methods work effectively without much parameter tuning on the ISP side.

- Efficient. We require that CableMon can detect whether there is a network fault or not in real time. This is because an ISP can deploy CableMon as a diagnosis tool in

	03/13/2019	04/09/19	06/25/19	07/15/19	08/15/19	Eight-month
MTR $<$ 18dB	24.95 %	25.45 %	27.16 %	27.07 %	27.38 %	26.15 %

Table 1: **The percentage of cable modems that need to be repaired if an ISP were to follow one of the CableLabs' recommendations.**

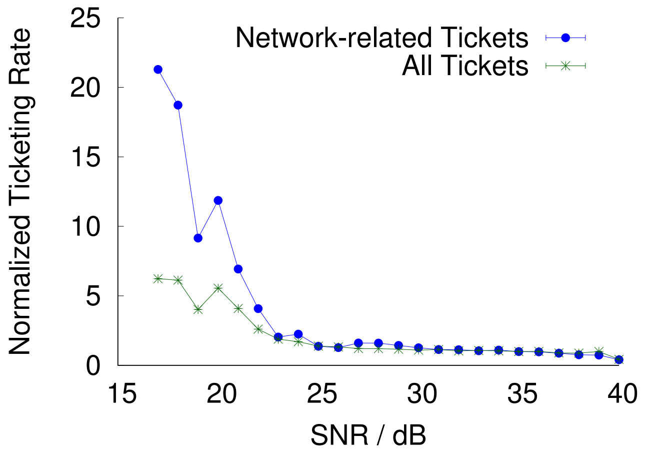


Figure 2: **This figure shows how the customer ticketing rate varies with the values of SNR. Ticketing rate tends to increase when SNR values are low.**

addition to using it for proactive network management. When an ISP receives a customer trouble call, it is often challenging to diagnose what has caused the customer's problem. An ISP can use CableMon to help diagnose whether the problem is caused by a network fault.

3.3 Design Rationale

To meet CableMon's design goals, we use customer trouble tickets as hints to train a customized classifier to detect network faults. We hypothesize that the occurrences of customer trouble tickets should correlate with those of network faults. When a customer-impacting fault occurs, some customers are likely to call the ISP to fix it. Each call creates a trouble ticket. If the values of PNM data can indicate network faults, then the values of PNM data should correlate with how frequently customer trouble tickets are created. In this paper, we define the average number of customer tickets created in a unit time as the *ticketing rate*.

To validate this hypothesis, we measure how ticketing rate changes with different values of a PNM metric. For a PNM metric m (e.g., SNR), we sort the values of m in an ascending order. Each pair of adjacent values define a bin b. For each bin b, we measure the number of tickets N_b that occur in the time periods where the value of m falls within the bin, and the total length of those time periods T_b. We then divide N_b by T_b to obtain the ticketing rate for bin b. We note that a PNM record is collected at discrete time points (roughly four hours apart in our datasets). We assume that a PNM value remains unchanged between its collection points.

As an example, we show how the ticketing rate varies with the values of SNR in Figure 2. We normalize this value by the baseline ticketing rate, which we obtain by dividing the total number of customer tickets in our dataset by the total collection period. The line marked by the legend "All Tickets" shows how the ticketing rate varies with the values of SNR if we consider all tickets; and the line marked by "Network-related Tickets" shows how the ticketing rate of network-related tickets varies with SNR. As can be seen, when the values of SNR are low, both the network-related ticketing rate and the all-ticket ticketing rate tend to increase, suggesting that low SNR values signal network faults.

In practice, customer tickets do not always indicate network faults. On the one hand, many customers may call an ISP for non-network related problems. The customer ticket data we obtain includes a ticket action field and a ticket description field, which provide the information on how an ISP deals with a ticket. We observe that nearly 25% of tickets are resolved via"Customer Education" or "Cancelled", suggesting that they are not caused by networking problems. On the other hand, customers may not report tickets when network outages indeed take place. In our ticket dataset, when an outage affects an entire region, all tickets caused by that outage are labeled as "part of primary," grouped and pointed to a primary ticket, which is a representative ticket of the outage. We manually checked an outage that affected more than 200 customers' PNM data and observed that only $\approx 6.1\%$ of the customers have a "part of primary" tickets and the rest $\approx 93.9\%$ of the customers did not report anything.

To reduce noise in tickets, we select a subset of customer tickets that are likely to be caused by network problems. We select the tickets based on both a ticket's action field and the ticket's description field. From the action field, we select tickets that lead to a "Dispatch" action. We assume that the tickets that caused an ISP to dispatch a technician are likely to be triggered by network-related problems. From the description field, we select tickets whose ticket description keywords suggest networking problems. Examples of such keywords include "Data Down", "Noisy Line" and "Slow Speed". In the rest of this paper, we refer to those selected tickets as "network-related tickets."

Figure 2 compares how the ticketing rate of network-related tickets and all tickets vary with SNR values. As can be seen, network-related tickets have higher ticketing rates when SNR is low, suggesting that the occurrences of those tickets are better indicators of network faults.

We note that according to the ISP who provided us the datasets, network-related tickets may also contain non-networking tickets due to human errors. A human operator who fills a ticket action or description field may make a mistake. And a technician may be dispatched when there is no network fault due to an erroneous diagnosis.

Challenges: A key question we need to answer is how to use customer tickets as hints for detecting network faults. Ideally, if a customer calls only when a network fault occurs, we could label the PNM records collected around the ticket creation time as abnormal, and apply supervised learning to learn the PNM thresholds that suggest a network fault. We have tried several such machine learning algorithms when we started this project, but found that that this approach did not work well with our datasets. First, customer calls are unreliable fault indicators. A customer may or may not call when there is a fault and vice versa. Second, PNM data contain noise. During a faulty period, some PNM metrics may occasionally show normal values due to the added noise. Similarly, even when there is no fault, some PNM metrics may show instantaneous abnormal values. Thus, if we use the tickets to label PNM data, it may introduce many false positives as well as many false negatives. We found it challenging to tune a machine learning algorithm with this labeling method. It is even harder to explain the results when we change a parameter. Next, we describe how we design CableMon to use a simple and customized classifier to address these challenges.

4 Design

In this section, we describe the design of CableMon. We first describe how we reduce the noise in customer tickets and the noise in PNM data. We then describe a customized classifier that aims to robustly classify PNM values as normal and abnormal despite the presence of noise. Finally, we describe how an ISP can use the classification results to detect network faults and to help diagnose a customer's trouble call.

4.1 Reducing Noise in PNM data

PNM data measure the instantaneous state of cable's RF channels and contain noise. An added noise may make a PNM metric take an abnormally low or high instantaneous value. To address this problem, we treat PNM data as time-series data and apply statistical models to smooth the noise and generate additional features for fault detection.

Table 2 summarizes all the statistical models we use to process PNM data. For each PNM metric collected at timestamp i with value V_i, we calculate its average, its weighted moving average (WMA), exponentially weighted moving average (EWMA), the difference between the current value and its WMA (WMA Diff), and its variance. We note that the average, WMA, WMA Diff, and variance values all require a window size as a hyper-parameter. Because we do not have any prior knowledge on how to set this parameter, we try a series of window sizes, ranging from *1 day* to *7 days*, incrementing by 1 day at each step. For the λ parameter required by EWMA, we vary the value of λ from *0.1* to *0.9*, incrementing by 0.1 at each step. For each PNM metric, we generate 37 time-series features. We apply this approach to all nine PNM metrics and totally generate 333 time-series features. We refer to them as time-series features.

Model	Equation
Average	$AVG_i = \frac{V_i+V_{i-1}+\cdots+V_{i-win+1}}{win}$
WMA	$WMA_i = \frac{win \cdot V_i+(win-1)\cdot V_{i-1}+\cdots+1\cdot V_{i-win+1}}{win\cdot(win-1)/2}$
EWMA	$EWMA_1 = V_1$ $EWMA_i = \lambda \cdot V_i + (1-\lambda)EWMA_{i-1}$
WMA Diff	$V_i - WMA_i$
Variance	$VAR_i = \frac{1}{win}\sum_{k=i-win+1}^{i}(V_k - AVG_i)^2$

Table 2: **This table summarizes the statistical models we use to generate the time-series features. (WMA: Weighted Moving Average, EWMA: Exponentially Weighted Moving Average.)**

4.2 Determining A Fault Detection Threshold

After we reduce noise in both the customer tickets and the PNM data, we aim to determine a threshold for each PNM metric that indicates network faults. We note that there is no explicit definition of what a network fault is. Instead, we choose to use the network conditions that are likely to cause a trouble call to approximate a network fault. With this approximation, we may not detect minor issues that do not warrant a trouble call. We argue that this design is advantageous, because it allows an ISP to prioritize its resources to fix the customer-impacting problems.

In the case of SNR, if we choose too high a value as a fault detection threshold, an ISP may become too proactive, fixing minor problems that many customers may not care, which we refer to as false positives. If we choose too low a value, an ISP may miss opportunities to proactively repair a problem before a customer calls, which we refer to as false negatives.

We aim to design an algorithm that minimizes both false positives and false negatives. From our investigation in § 3.3, we see that different values of a PNM metric have different likelihood to concur with a trouble ticket. Inspired by this observation, we use the ticketing rate as a metric to help choose a fault detection threshold. Our intuition is that the customer ticketing rate during a faulty period should be higher than a normal period when there is no fault. Therefore, for each feature f generated from a PNM metric, we determine a threshold value thr_f such that thr_f maximizes the ratio between the ticketing rate in the time periods when a network fault exists and the time periods when there is no fault. We refer to this ratio as the ticketing rate ratio.

Specifically, we search through the range of values of a feature f in small steps. At each step s, we consider the value of the feature $f_s \in [f_{min}, f_{max}]$, as a candidate for the threshold.

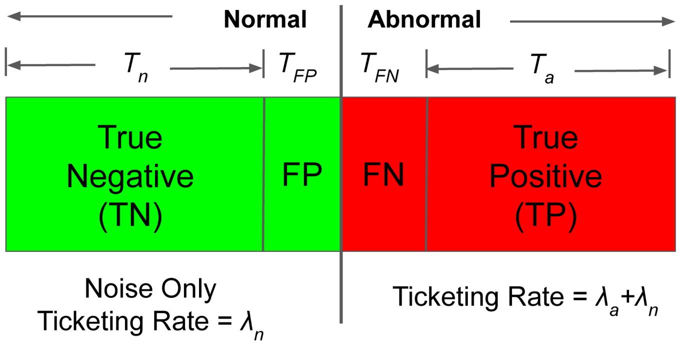


Figure 3: **Analysis of ticketing rate ratio.**

We then compare the value of f at a PNM data collection point with f_s, and label the collection time period as abnormal or normal, based on whether the value of f is below or above the candidate threshold value f_s. For some features such as the average SNR, below the threshold is abnormal. For other features, the opposite is true. After determining each collection period as normal or abnormal, we count the number of network-related tickets occurred in the normal and abnormal periods respectively and divide them by the normal and abnormal time periods determined by f_s. We then compute the ticketing rate ratio: $TRR(f_s)$. The threshold value thr_f is chosen as the value of f_s that maximizes the ticketing rate ratio $TRR(f_s)$.

We also note that for features following a normal distribution such as Rx Power, we choose to use two threshold values to determine whether a collection period is normal or abnormal. The pseudo code can be found at §A.

We now explain why choosing a threshold that maximizes the ticketing rate ratio may help minimize the false positives and false negatives. The entire time line can be divided into two subspaces: the normal (no fault) and the abnormal (with fault) periods. Ideally, the normal sub-space should not receive any trouble ticket. In practice, there is always a ticketing noise. We assume a uniformly distributed ticketing noise with the rate λ_n spreads the whole space. Similarly, we assume an additional uniformly distributed ticketing rate that occurs only in the abnormal sub-space and denote it as λ_a.

A threshold value thr_f of a feature also divides the time line into two subspaces: normal and abnormal. The first subspace includes a true negative part T_n and a false negative part T_{FN}, where an abnormal period is erroneously considered as normal. The second subspace includes a true positive part T_a and a false positive part T_{FP}, where a normal period is considered abnormal. The ticketing rate ratio determined by the threshold thr_f can be computed as follows:

$$D(T_n, T_{FP}, T_{FN}, T_a) = \frac{\frac{\lambda_n T_{FP} + (\lambda_a + \lambda_n) T_a}{T_a + T_{FP}}}{\frac{\lambda_n T_n + (\lambda_a + \lambda_n) T_{FN}}{T_n + T_{FN}}}$$

Both the numerator and denominator can be regarded as a weighted average of λ_n and $\lambda_a + \lambda_n$, with the time period lengths as the weights. Because $\lambda_a + \lambda_n > \lambda_n$ always holds, we can show that the derivatives of D over the false positives T_{FP} and the false negatives T_{FN} are non-increasing:

$$\frac{\partial D}{\partial T_{FP}} < 0 \text{ and } \frac{\partial D}{\partial T_{FN}} < 0$$

Therefore, because T_{FP} and T_{FN} are non-negative, the ticket rate ratio is maximized when both false positives and false negatives are zero:

$$D_{max} = \lim_{\substack{T_{FP} \to 0 \\ T_{FN} \to 0}} D = \frac{\lambda_a}{\lambda_n} + 1$$

4.3 Feature Selection

We have a total of more than three hundred time-series features and it is unlikely they are all useful indicators of network faults. To find the relevant features, we only select the features with high ticketing rate ratios from each PNM metric. Specifically, among the same type of features derived from a PNM metric with different hyperparameters, we choose the one with the highest ticketing rate ratio as the representative feature. For each representative feature derived from the same PNM value, we choose the top two with the highest ticketing rate ratios. Finally, among the remaining candidates, we choose the top N features that have the highest ticketing rate ratios. We determine the number of features N based on the desired ticketing rate ratios, ticket prediction accuracy, and ticket coverage as we soon describe in § 5.1.

Features	Ticketing Rate Ratio
snr-var-2	14.49
uncorrected-var-1	7.66
rxpower-wma-diff-4	5.31
t3timeouts-wma-diff-1	4.93
t4timeouts-var-1	4.18

Table 3: **Top 5 features and their ticketing rate ratio.**

Table 3 shows the top five features we used and their ticketing rate ratios calculated from our training sets (Section 5.2). The name of each feature consists of the raw PNM metric, the statistical model we apply to the metric, and the parameter. For example, the *snr-var-2* means the variance of SNR with a *2-days* window size. We note that all features have a high ticketing rate ratio and we expect them to effectively detect network faults.

4.4 Combining Different Features

Different PNM features may detect different types of network faults. Therefore, we build the final classifier by combining the detection results of all selected features. As long as one selected feature considers a PNM collection period abnormal, we classify the collection period as abnormal. For each

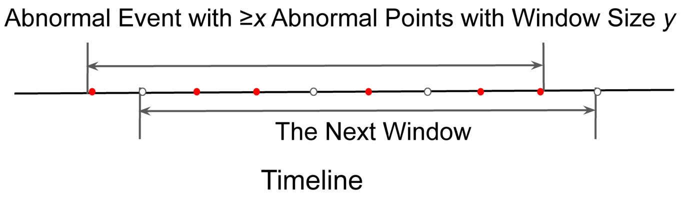


Figure 4: **This figure explains the sliding window algorithm. When the number of abnormal points within a sliding window exceeds a threshold, the window is considered to be abnormal. An abnormal event is given by merging the abnormal windows.**

selected feature, we have already chosen a threshold that maximizes the ticketing rate ratio. Therefore, we expect that combining the results of all selected features will also provide a high ticketing rate ratio. We evaluate the results of our classifier in § 5.1.

4.5 ISP Deployment

An ISP can use CableMon in two ways: proactive network maintenance for predicted trouble tickets and diagnosing the root cause of a trouble ticket when receiving a call. In this section, we describe the algorithms for an ISP to decide when to send out a repair technician proactively and how to diagnose the root cause.

CableMon's classifier can monitor an ISP's network continuously. It can output a normal and abnormal decision when a PNM record is collected from a customer's modem. However, due to the existence of noise and the intermittent nature of some faults, if an ISP makes a dispatch decision whenever it observes an abnormal PNM data point, it may lead to many false positives. To address this problem, we design a sliding window algorithm for an ISP to make a dispatch decision. The high-level idea of this algorithm is that an ISP should only dispatch a technician after a fault persists.

Figure 4 explains this algorithm. The algorithm takes two parameters: y and x, where y is the size of the window, and x is the number of abnormal data points detected in the window. When an ISP collects a new PNM record, it looks back to a window size y of collection points. If x out of y data points are considered as abnormal, then the ISP should dispatch a technician to examine and repair the network.

An ISP can determine the parameters x and y based on the false positives and false negatives it is willing to tolerate. The ISP can estimate the values of false positives and false negatives from its historic PNM data and ticket data. Therefore, choosing those parameters only requires an ISP to train CableMon using its own PNM and ticket data and does not require tuning. In § 5.1, we use our datasets to show how an ISP can effectively choose the parameters x and y.

Similarly, an ISP can use CableMon to help diagnose the root cause of a call. When it receives a trouble call, if the customer complains about a performance problem, and the ISP sees that in the past collection window of size y, there exists x abnormal collection points, the ISP can conclude that the trouble is likely to be caused by a network problem.

5 Evaluation

In this section, we describe how we evaluate CableMon's performance.

5.1 Establishing Evaluation Metrics

Ideally, we would like to deploy CableMon on a real cable ISP and measure how it reduces the number of trouble tickets over a long term. It is our future work to conduct such a real-world experiment. In this work, we aim to estimate how many trouble tickets CableMon would reduce were it deployed on our collaborating ISP.

To do so, we emulate the sliding window algorithm described in § 4.5 using our test dataset. We start from the beginning of the test dataset. If there are x abnormal points detected in a window size of y, we mark it as the beginning of a fault. We then move the window to the next data point. When the number of abnormal points falls below x, we mark it as the end of a fault. If there is a trouble ticket occurred during a fault, we consider this fault detection as a true fault. We note that if we detect a fault simultaneously within multiple customers, as long as one customer reports a ticket, we consider it a true fault. We assume that if an ISP dispatched a repair technician when it detected the onset of the fault, it could have avoided the trouble ticket. We define *ticket prediction accuracy* as the number of true faults divided by the total number of detected faults. We define *ticket coverage* as the number of trouble tickets occurred during a detected fault divided by the total number of network-related trouble tickets.

It is not sufficient to use only ticket prediction accuracy and ticket coverage to gauge CableMon's performance. This is because if CableMon detects the entire time period that spans the test dataset as a faulty period, it will achieve 100% ticket coverage and ticket prediction accuracy. To avoid this pitfall, we also use the normalized ticketing rate, which is defined as the ticketing rate in all faculty periods normalized by the ticketing rate of the time period that spans the test dataset. If CableMon erroneously detects the entire time period as abnormal, it will achieve a low normalized ticketing rate close to 1.

How an ISP chooses the sliding window parameters: In practice, an ISP can use a training set to determine the threshold values of CableMon's classifier. It can use the ticket prediction accuracy, ticket coverage, and the normalized ticketing rate obtained from a validation set to choose the combination of the sliding window parameters.

We show an example in Figure 5. In this example, we choose a window size of 12 data points (y = 12), which is roughly two days long. We then measure the ticket prediction accuracy, ticket coverage, and the normalized ticketing rate

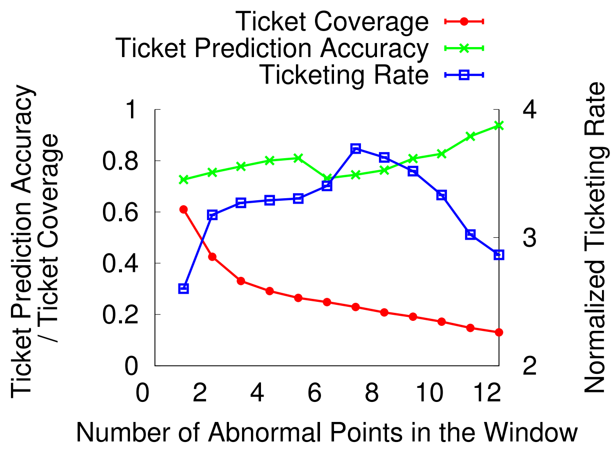


Figure 5: **This figure shows the ticket detection accuracy, the ticket coverage, and the normalized ticketing rate of the sliding window algorithm with different parameters.**

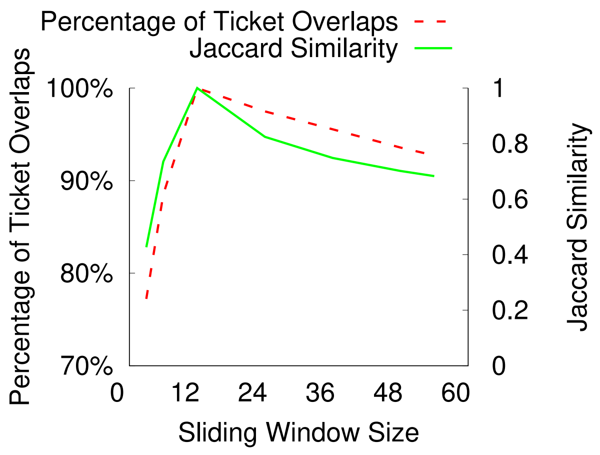


Figure 6: **This figure shows what percentage of tickets detected by different window sizes overlap with those detected by a window size of 12 and the Jaccard similarity between the faulty periods detected by different window sizes and those detected by a window size of 12.**

when the number of abnormal points x varies from 0 to 12. As can be seen, when x is around 8, the sliding window algorithm achieves a high normalized ticketing rate, a relatively high ticket prediction accuracy 80%, and a ticket coverage around 20%. Since avoiding false dispatches is more important than predicting all trouble tickets, an ISP can choose (8, 12) as its sliding window parameters for fault detection.

We have tried different sizes of the sliding window, ranging from one to 60 data points. For each window size, we use the above method to choose the parameter x such that both the ticket prediction accuracy and the normalized ticketing rate are high, and the ticket coverage is above a minimum threshold 15%. We compare the tickets and the faulty periods detected by different window parameters. We use the Jaccard similarity metric [25] to measure the overlaps of faulty periods detected by different window parameters. As can be seen in Figure 6, 90% of the tickets detected by windows larger than 12 overlap; and the faulty periods detected by them have a Jaccard similarity larger than 60%. This result suggests that different window parameters are likely to detect the same sets of faults, and the performance of CableMon is not sensitive to the window parameters.

5.2 Experiment Setup

After we establish the evaluation metric, we train and evaluate CableMon on a 50-machine Linux cluster with 40 $\sim$ 512 GB RAM and 8 $\sim$ 48 core running Ubuntu 18.04. CableMon is trained on five-month data from 01/06/2019 to 05/30/2019 and tested with three-month data from 06/01/2019 to 08/31/2019.

Comparing with the existing work: We compare CableMon's performance with two existing methods. One is from our collaborating ISP, AnonISP, which uses a visualization tool that colors different ranges of PNM values for an operator to manually monitor its networks' conditions. AnonISP's tool has two manually configured thresholds for several raw PNM values and therefore has three fault indication levels: normal (green), marginal (yellow), and abnormal (red). We compare AnonISP's tool against CableMon with these thresholds and regard both yellow and red levels as network fault, as the ISP's experts usually do.

Another tool from industry uses Comcast's scoreboard method [22]. Comcast is considered as the leading company in the area of PNM research. They developed a method that compares a PNM metric to a threshold and assigns a score to each comparison result. If the sum of the comparison scores exceeds a threshold value, then the method considers there is a fault in the network.

Since both AnonISP and Comcast's tool detects a fault using a single PNM data record, we apply the sliding window algorithm to both tools for a fair comparison.

Comparing with Machine Learning Techniques: We also compare the performance of CableMon with three classical machine learning algorithms: Decision Tree (DT) [34], Random Forest (RF) [4] and Support Vector Machine (SVM) [14]. Since these algorithms require labeled data, we label the PNM data with tickets. Each ticket has a creation time and a closed time. We label the PNM data collected between this time interval as positive samples and other data as negative samples. We generate 47,518 positive samples and the same number of abnormal samples as our training set to train the machine learning models and evaluate them with the same evaluation metrics.

Table 4 shows the ticket prediction accuracy, the ticket coverage, and the normalized ticketing rate of different methods. As can be seen, CableMon achieves the highest ticket prediction accuracy and the highest normalized ticketing rate

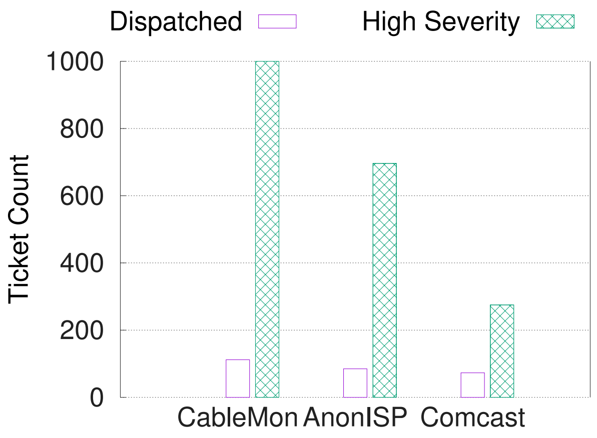


Figure 7: **This figure shows the number of different types of tickets detected by different methods.**

among all methods. Its ticket coverage is lower than that of AnonISP. However, this is because AnonISP detects too many false faults, as shown by its low ticket prediction accuracy. We note that all three machine learning algorithms require a long training time, as each has multiple parameters to tune. The results we present here are the best ones after many rounds of tuning. When we started this project, we started with those algorithms, but abandoned them due to the challenges to tune them and to explain the results when certain parameters are changed.

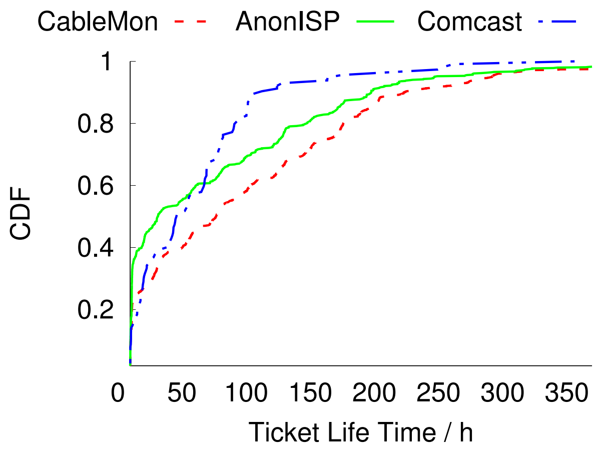
Methods	Ticket Prediction Accuracy	Ticket Coverage	Normalized Ticketing Rate
CableMon	81.92%	22.99%	3.55
Decision Tree	68.93%	15.53%	2.52
SVM	75.64%	12.54%	2.02
Random Forest	73.14%	14.21%	2.24
Comcast	23.48%	2.21%	1.18
AnonISP's tool	10.04%	25.13%	0.98

Table 4: **Performance of different methods**

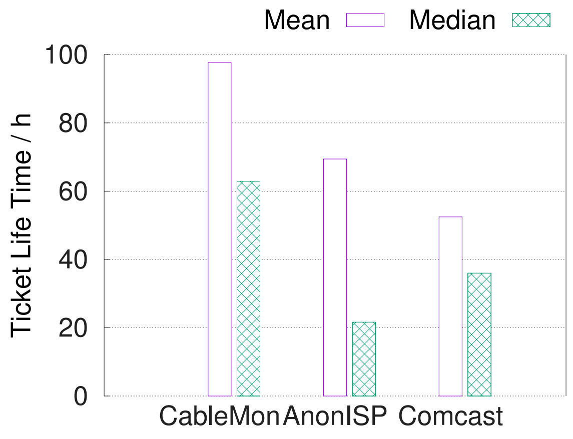
5.3 Detected Tickets Statistics

To further analyze the detected tickets, we examine the tickets detected by CableMon and existing ISP tools according to the ticket action and description fields. We omit the results of the machine learning algorithms for clarity. The characteristics of the tickets detected by those algorithms are similar to those of CableMon, but they have lower ticket detection accuracy and coverage. Figure 7 shows the number of different types of tickets detected by different methods. As can be seen, CableMon can detect a significantly more number of dispatched and high severity tickets than the two existing ISP tools.

Figure 8(a) shows the distribution of a detected ticket's life time, and figure 8(b) shows the average and median life time



(a) CDF



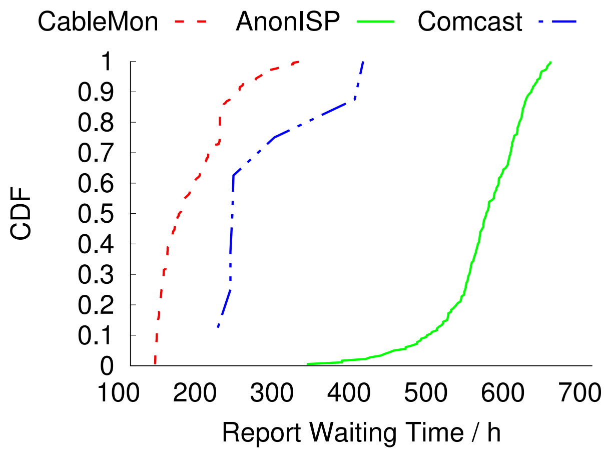
(b) Mean and Median

Figure 8: **The figures show the CDF, mean, and median of the life time of tickets predicted by different methods. A longer ticket life time indicates that the problem that triggered the ticket takes a longer time to fix.**

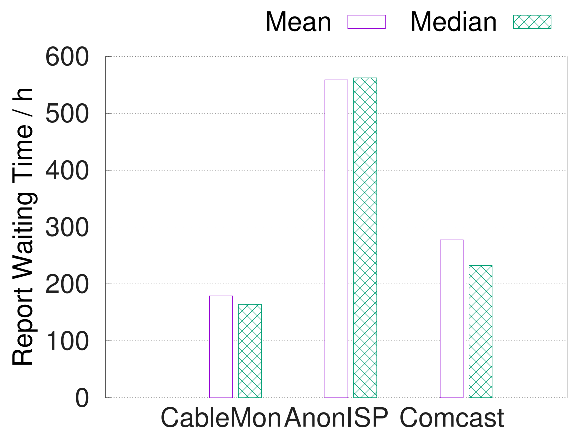
of a detected ticket. A ticket's life time is defined as the time between a ticket is created to the time a ticket is closed. As can be seen, the tickets detected by CableMon have longer life times, suggesting that CableMon detects the problems that take longer to resolve.

We also measure the time elapsed from when a fault is detected to when a ticket is created. We refer to this time as "Report Waiting Time." Figure 9(a) shows the cumulative distribution of the report waiting time of different methods, and figure 9(b) shows the average and median report waiting time of different methods. As can be seen, CableMon's report waiting time is also significantly shorter that that of other methods, indicating that its detected faults lead to customer trouble tickets faster than those detected by other methods.

Finally, we measure the distribution of a fault detected by different methods. Figure 10 shows the PDF of the length of a fault detected by different methods. As can be seen,



(a) CDF



(b) Mean and Median

Figure 9: **The figures show the CDF, mean, and median of the report waiting time of tickets predicted by different methods. A shorter report waiting time indicates that the problem triggered by the ticket is more urgent.**

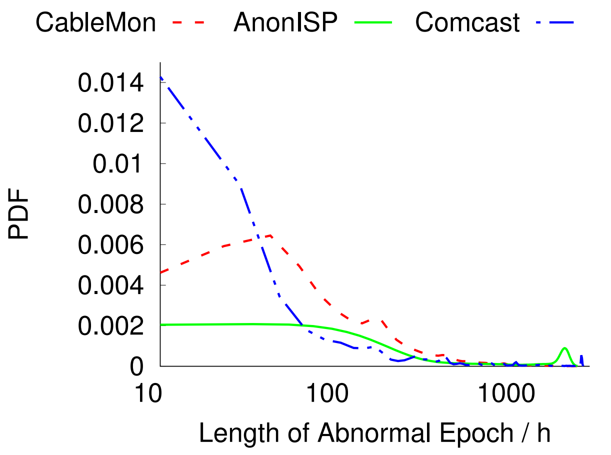


Figure 10: **This figure shows the PDF of the length of a detected fault.**

CableMon detected faults tend to last a moderate period of time. The highest probability density is slightly less than 100 hours (roughly four days). Comcast's tool detects many faults that last less than one day, shorter than what a typical network repair action takes. This result suggests that many of the detected faults could be false positives. The faults detected by AnonISP's tool have a wide range of life span, from very short faults to very long faults (>500 hours), which are outside the normal range of repair actions. Again, this result suggests that many of the detected faults could be false positives.

6 Related Works

Previous work measured the reliability of broadband networks. The Federal Communications Commission launched the Measuring Broadband America (MBA) project [10] since 2010. Bischof et al. [3] showed that poor reliability will heavily affect user traffic demand. Padmanabhan et al. [28] demonstrated that the outages of broadband networks tend to happen under bad weather conditions. Baltrunas et al. [2] also measured the reliability of mobile broadband networks.

Network fault diagnosis has attracted much attention from the community for a long time. Many approaches from industry, especially the cable industry, set manual thresholds for certain measured metrics to detect network outages. Amazon [11] used a fixed threshold to monitor the condition for its cloud services. Zhuo et al. [41] treated packet loss as a fault indicator and showed the correlation between Tx/Rx Power and packet loss rate. They again use manually set thresholds to detect network faults. Lakhina et al. [21] proposed the first framework that applied Principal Component Analysis (PCA) to reduce the dimension of network traffic metrics. Huang et al. [16] showed that Lakhina's framework works well with a limited number of network devices, but has performance issues on larger datasets. Moreover, Ringberg et al. [35] pointed out that using PCA for traffic anomaly detection is much more tricky than it appears. Besides PCA, many other statistical models are applied to network anomaly detection. Gu et al. [13] measured the relative entropy of the metrics and compared them to the baseline. Subspace [26] is introduced to deal with high-dimensional and noisy network monitoring data. Kai et al. [19] used Expectation–Maximization (EM) algorithm to estimate the parameters of their model and obtain the upper or lower bound of the common metric values. Independent Component Analysis [30], Markov Modulated Process [32], and Recurrence Quantification Analysis [29] are also introduced to find the anomaly points in time series data. These methods aim to detect sudden changes in data. Differently, CableMon uses customer ticket as hints to label the input data and uses the ticketing rate ratio to select relevant features.

Recently, machine learning has been used for network anomaly detection. Leung et al. [24] designed a networking anomaly detection system using a density-based clustering algorithm, which obtained the accuracy as 97.3%. Dean et al. [9] presented an Unsupervised Behavior Learning framework based on the clustering algorithm. However, cluster-based approaches do not work well with sparse data, which is the case of our PNM data where abnormal events are rare. Sung et al. [37] deployed Support Vector Machines (SVMs) to estimate the actual crucial features. According to our evaluation, SVMs do not perform as well as CableMon. Liu et al. [27] adopted more than twenty statistics models to obtain more features from the original data. They used all generated features in Random Forest and achieved high accuracy and effectiveness. However, they still require manual labeling to train the Random Forest model. PreFix [39] predicts switch failures with high precision and recall. However, it also requires significant manual efforts for labeling, while our work does not. Pan et al. [31] also used the tickets as the hints to select potential network faults. However, they still asked experts to manually label network faults and use this labelled data to train a Decision Tree model. In contrast, CableMon does not use any manual label.

Previous researches have also focused on processing customer report tickets. LOTUS [38] deploys Natural Language Processing (NLP) techniques to understand the tickets. Potharaju et al. [33] built a system that automatically processes the raw text of tickets to infer the networking faults and find out the resolution actions. Jin et al. [18] studied the tickets in cellular networks and categorized the types of customer trouble tickets. Chen et al. [8] and Hsu et al. [15] use both customer trouble tickets and social media postings to determine network outages. This work combines an ISP's customer trouble tickets and PNM data to infer network faults.

7 Discussion

CableMon uses customer trouble tickets as network fault indicators to build a classifier without manual labeling. We plan to focus on the following directions to improve the performance of CableMon:

- When there lacks a large set of labeled data, semi-supervised learning [40] combines a small set of labeled data and a large set of unlabeled data to improve classification accuracy. We plan to investigate whether semi-supervised learning approach as well as other machine learning methods such as deep learning can improve the performance of CableMon.

- Presently, we use network-related tickets to train the classifier. We have discovered that customers tend to report tickets at weekdays rather than on weekends and during the day rather than at night. From this pattern, one may infer that if a customer reports a ticket at an "atypical" time, it is more likely to indicate a customer-impacting problem. If we place a higher weight for such "outlier" tickets in a classification algorithm, we may increase both the ticket prediction accuracy and coverage.

- ISPs desire to differentiate failures that affect a group of customers from those that affect a single customer. We refer to faults that affect multiple customers as "maintenance issues." If there is a maintenance issue, it is also desirable to locate the place where this issue has happened. It is possible to infer maintenance issues by clustering customers' PNM data, and to infer the location of a maintenance issue by combining the geographical location of each modem with the topology of HFC network. It is our future work to study these problems.

- When detecting network faults, CableMon outputs whether there is an abnormal event and how long it exists. It is desirable to rank the severity of abnormal events so that an ISP can prioritize its repair actions. It is our future work to explore such ranking algorithms.

8 Conclusion

Cable broadband networks are widely deployed all around U.S. and serve millions of U.S. households. However, cable networks have poor reliability. Although the cable industry has developed a proactive network maintenance (PNM) platform to address this issue, cable ISPs have not fully utilized the collected data to proactively detect and fix network faults. Existing approaches rely on instantaneous PNM metrics with manually set thresholds for fault detection and can introduce an unacceptably high false positive rate. We design CableMon, a system that learns the fault detection criteria from customer trouble tickets. Our design overcomes the noise from both PNM data and customer trouble tickets and achieves a nearly four times higher ticket prediction accuracy than the existing tools in the public domain. This is a first step toward enabling an ISP to use PNM data to proactively repair a failure before a customer calls and to diagnose whether a customer trouble call is caused by a network fault.

Acknowledgment

The authors would like to thank the NSDI Shepherd Vyas Sekar and anonymous reviewers for their valuable feedback. This work is supported in part by an NSF award CNS-1910867.

References

[1] History of Cable. https://www.calcable.org/learn/history-of-cable/, 2018.

[2] Dziugas Baltrunas, Ahmed Elmokashfi, and Amund Kvalbein. Measuring the Reliability of Mobile Broadband Networks. In *ACM IMC*, 2014.

[3] Zachary S Bischof, Fabian E Bustamante, and Nick Feamster. Characterizing and Improving the Reliability of Broadband Internet Access. In *46th Research Conference on Communication, Information and Internet Policy (TPRC)*, 2018.

[4] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[5] CableLabs. Data Over Cable Service Interface Specifications DOCSIS 3.0 - Operations Support System Interface Specification. 2007.

[6] CableLabs. Cable Broadband Technology Gigabit Evolution. https://www.cablelabs.com/insights/cable-broadband-technology-gigabit-evolution/, 2016.

[7] DOCSIS CableLabs. Best Practices and Guidelines, PNM Best Practices: HFC Networks (DOCSIS 3.0). Technical report, CM-GL-PNMP-V03-160725, 2016.

[8] Yi-Chao Chen, Gene Moo Lee, Nick Duffield, Lili Qiu, and Jia Wang. Event Detection Using Customer Care Calls. In *IEEE INFOCOM*, 2013.

[9] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems. In *ACM International Conference on Autonomic Computing*, 2012.

[10] Federal Communications Commission (FCC). In the Matter of Reliability and Continuity of Communication Networks. PS Docket 11-60, 2011.

[11] Filippo Lorenzo Ferraris, Davide Franceschelli, Mario Pio Gioiosa, Donato Lucia, Danilo Ardagna, Elisabetta Di Nitto, and Tabassum Sharif. Evaluating the Auto Scaling Performance of Flexiscale and Amazon EC2 Clouds. In *IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012.

[12] Sarthak Grover, Mi Seon Park, Srikanth Sundaresan, Sam Burnett, Hyojoon Kim, Bharath Ravi, and Nick Feamster. Peeking Behind the NAT: An Empirical Study of Home Networks. In *ACM IMC*, 2013.

[13] Yu Gu, Andrew McCallum, and Don Towsley. Detecting Anomalies in Network Traffic Using Maximum Entropy Estimation. In *ACM IMC*, 2005.

[14] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support Vector Machines. *IEEE Intelligent Systems and their applications*, 13(4), 1998.

[15] Wenling Hsu, Guy Jacobsen, Yu Jin, and Ann Skudlark. Using Social Media Data to Understand Mobile Customer Experience and Behavior. In *European Regional Conference of the International Telecommunications Society*, 2011.

[16] Ling Huang, XuanLong Nguyen, Minos Garofalakis, Michael I Jordan, Anthony Joseph, and Nina Taft. In-Network PCA and Anomaly Detection. In *Advances in Neural Information Processing Systems*, 2007.

[17] David Hunter and Tom Williams. Improved Customer Service Through Intermittent Detection. In *SCTE Cable-Tec Expo*, 2015.

[18] Yu Jin, Nick Duffield, Alexandre Gerber, Patrick Haffner, Wen-Ling Hsu, Guy Jacobson, Subhabrata Sen, Shobha Venkataraman, and Zhi-Li Zhang. Making Sense of Customer Tickets in Cellular Networks. In *IEEE INFOCOM*, 2011.

[19] Huang Kai, Qi Zhengwei, and Liu Bo. Network Anomaly Detection Based on Statistical Approach and Time Series Analysis. In *IEEE International Conference on Advanced Information Networking and Applications Workshops*, 2009.

[20] D Richard Kuhn. Sources of Failure in the Public Switched Telephone Network. *Computer*, 30(4):31–36, 1997.

[21] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing Network-Wide Traffic Anomalies. In *ACM SIGCOMM Computer Communication Review*, 2004.

[22] Bryan Thomas Larry Wolcott, John Heslip and Robert Gonsalves. A Comprehensive Case Study of Proactive Network Maintenance. In *SCTE Cable-Tec Expo*, 2016.

[23] William Lehr, Mikko Heikkinen, David Clark, and Steven Bauer. Assessing Broadband Reliability: Measurement and Policy Challenges. *Research Conference on Communication, Information and Internet Policy (TPRC)*, 2011.

[24] Kingsly Leung and Christopher Leckie. Unsupervised Anomaly Detection in Network Intrusion Detection Using Clusters. In *Twenty-Eighth Australasian Computer Science Conference*, 2005.

[25] Michael Levandowsky and David Winter. Distance Between Sets. *Nature*, 234(5323):34, 1971.

[26] Xin Li, Fang Bian, Mark Crovella, Christophe Diot, Ramesh Govindan, Gianluca Iannaccone, and Anukool Lakhina. Detection and Identification of Network Anomalies Using Sketch Subspaces. In *ACM IMC*, 2006.

[27] Dapeng Liu, Youjian Zhao, Haowen Xu, Yongqian Sun, Dan Pei, Jiao Luo, Xiaowei Jing, and Mei Feng. Opprentice: Towards Practical and Automatic Anomaly Detection through Machine Learning. In *ACM IMC*, 2015.

[28] Ramakrishna Padmanabhan, Aaron Schulman, Dave Levin, and Neil Spring. Residential links under the weather. In *ACM SIGCOMM*. 2019.

[29] Francesco Palmieri and Ugo Fiore. Network Anomaly Detection Through Nonlinear Analysis. *Computers & Security*, 29(7):737–755, 2010.

[30] Francesco Palmieri, Ugo Fiore, and Aniello Castiglione. A Distributed Approach to Network Anomaly Detection Based on Independent Component Analysis. *Concurrency and Computation: Practice and Experience*, 26(5):1113–1129, 2014.

[31] Lujia Pan, Jianfeng Zhang, Patrick PC Lee, Hong Cheng, Cheng He, Caifeng He, and Keli Zhang. An Intelligent Customer Care Assistant System for Large-Scale Cellular Network Diagnosis. In *ACM International Conference on Knowledge Discovery and Data Mining*, 2017.

[32] Ioannis Ch Paschalidis and Georgios Smaragdakis. Spatio-Temporal Network Anomaly Detection by Assessing Deviations of Empirical Measures. *IEEE/ACM Transactions on Networking (TON)*, 17(3):685–697, 2009.

[33] Rahul Potharaju, Navendu Jain, and Cristina Nita-Rotaru. Juggling the Jigsaw: Towards Automated Problem Inference from Network Trouble Tickets. In *USENIX/ACM NSDI*, 2013.

[34] J Ross Quinlan. *C4. 5: Programs for Machine Learning*. Elsevier, 2014.

[35] Haakon Ringberg, Augustin Soule, Jennifer Rexford, and Christophe Diot. Sensitivity of PCA for Traffic Anomaly Detection. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2007.

[36] W Sawyer. Management Information Base for Data Over Cable Service Interface Specification (DOCSIS) Cable Modem Termination Systems for Subscriber Management. *RFC 4036*, 2005.

[37] Andrew H Sung and Srinivas Mukkamala. Identifying Important Features for Intrusion Detection using Support Vector Machines and Neural Networks. In *IEEE Symposium on Applications and the Internet.*, 2003.

[38] Shobha Venkataraman and Jia Wang. Assessing the Impact of Network Events with User Feedback. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 2018.

[39] Shenglin Zhang, Ying Liu, Weibin Meng, Zhiling Luo, Jiahao Bu, Sen Yang, Peixian Liang, Dan Pei, Jun Xu, Yuzhi Zhang, Yu Chen, Hui Dong, Xianping Qu, and Lei Song. Prefix: Switch Failure Prediction in Datacenter Networks. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[40] Xiaojin Zhu and Andrew B Goldberg. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130, 2009.

[41] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *ACM SIGCOMM*, 2017.

A Pseudo-code of Determining Thresholds

Algorithm 1 Threshold Determining

```
1: $\mathcal{P}_{label} \leftarrow$ Set of all data points associated with tickets
2: $\mathcal{P} \leftarrow$ Set of all data points
3: **if** one threshold **then**
4:     $\mathcal{V} \leftarrow$ Set of all distinct values
5:     **for** each $v \in \mathcal{V}$ **do**
6:         $N_l \leftarrow \{p | p \in \mathcal{P}_{label}, p.value \le v\}$
7:         $T_l \leftarrow \sum_{p \in \mathcal{P}, p.value \le v} p.time$
8:         $N_r \leftarrow \{p | p \in \mathcal{P}_{label}, p.value > v\}$
9:         $T_r \leftarrow \sum_{p \in \mathcal{P}, p.value > v} p.time$
10:        $TRR_v \leftarrow \max(\frac{|N_l|/T_l}{|N_r|/T_r}, \frac{|N_r|/T_r}{|N_l|/T_l})$
11:        $thr_m \leftarrow \arg\max TRR_v$
12:    **return** $thr_m$
13: **else**
14:    $\mathcal{A} \leftarrow$ Sorted array of all data points
15:    $\mathcal{B} \leftarrow$ Binning $\mathcal{A}$
16:    $\mathcal{V} \leftarrow$ Set of maximum value in each bin $b \in \mathcal{B}$
17:    **for** each $v_l \in \mathcal{V}$ **do**
18:        **for** each $v_r \in \mathcal{V}$ **do**
19:            $N_n \leftarrow \{p | p \in \mathcal{P}_{label}, v_l \le p.value \le v_r\}$
20:            $T_n \leftarrow \sum_{p \in \mathcal{P}, v_l \le p.value \le v_r} p.time$
21:            $N_a \leftarrow \{p | p \in \mathcal{P}_{label}, p \notin N_n\}$
22:            $T_a \leftarrow \sum_{p \in \mathcal{P}, p.value > v_r | p.value < v_l} p.time$
23:            $TRR_{(v_l, v_r)} \leftarrow \frac{|N_a|/T_a}{|N_n|/T_n}$
24:            $thr_l, thr_r \leftarrow \arg\max TRR_{(v_l, v_r)}$
25:    **return** $thr_l, thr_r$
```

Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives

Tamás Lévai[1,2], Felicián Németh[1], Barath Raghavan[2], and Gábor Rétvári[3,4]
[1]*Budapest University of Technology and Economics*
[2]*University of Southern California*
[3]*MTA-BME Information Systems Research Group*
[4]*Ericsson Research, Hungary*

Abstract

Data flow graphs are a popular program representation in machine learning, big data analytics, signal processing, and, increasingly, networking, where graph nodes correspond to processing primitives and graph edges describe control flow. To improve CPU cache locality and exploit data-level parallelism, nodes usually process data in batches. Unfortunately, as batches are split across dozens or hundreds of parallel processing pathways through the graph they tend to fragment into many small chunks, leading to a loss of batch efficiency.

We present Batchy, a scheduler for run-to-completion packet processing engines, which uses controlled queuing to efficiently reconstruct fragmented batches in accordance with strict service-level objectives (SLOs). Batchy comprises a runtime profiler to quantify batch-processing gain on different processing functions, an analytical model to fine-tune queue backlogs, a new queuing abstraction to realize this model in run-to-completion execution, and a one-step receding horizon controller that adjusts backlogs across the pipeline. We present extensive experiments on five networking use cases taken from an official 5G benchmark suite to show that Batchy provides 2–3× the performance of prior work while accurately satisfying delay SLOs.

1 Introduction

One near-universal technique to improve the performance of software packet processing engines is batching: collect multiple packets into a single burst and perform the same operation on all the packets in one shot. Processing packets in batches is much more efficient than processing a single packet at a time, thanks to amortizing one-time operational overhead, optimizing CPU cache usage, and enabling loop unrolling and SIMD optimizations [7,8,11,22,26]. Batch-processing alone often yields a 2–5× performance boost. Fig. 1 presents a series of micro-benchmarks we performed in BESS [14] and FastClick [3], two popular software switches, showing that executing an ACL or a NAT function is up to 4 times

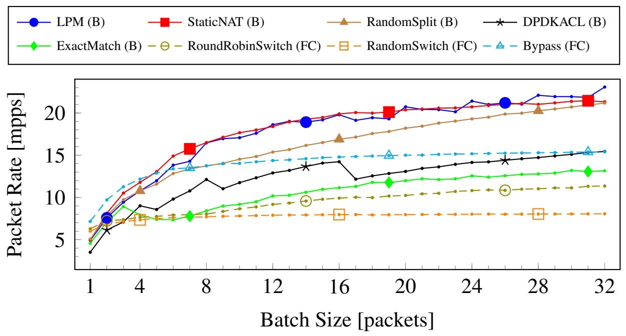


Figure 1: Maximum packet rate (in millions of packets per second, mpps) over different packet processing micro-benchmarks in BESS [14] (marked with B) and in FastClick [3] (FC) when varying the input batch size.

as efficient on batches containing 32 packets compared to single-packet batches. Prior studies provided similar batch-processing profiles in VPP [2,22], [27, Fig. 3]. Accordingly, batching is used in essentially all software switches (VPP [2, 27], BESS [14], FastClick [3], NetBricks [36], PacketShader [15], and ESwitch [30]), high-performance OS network stacks and dataplanes [3,6,8,11], user-space I/O libraries [1,16], and Network Function Virtualization platforms [19,21,42,45,50].

Unfortunately, even if the packet processing engine receives packets in bursts [1,2,6,27,29,41,50], batches tend to break up as they progress through the pipeline [20]. Such *batch-fragmentation* may happen in a multi-protocol network stack, where packet batches are broken into smaller per-protocol batches to be processed by the appropriate protocol modules (e.g., pure Ethernet, IPv4, IPv6, unicast/multicast, MPLS, etc.) [8,11]; by a load-balancer that splits a large batch into smaller batches to be sent to different backends/CPUs for processing [19]; in an OpenFlow/P4 match-action pipeline where different flow table entries may appoint different next-stage flow tables for different packets in a batch [30,38], or in essentially any packet processing engine along the branches of the data flow graph (splitters) [3,14,16,31,47]. In fact, any operation that involves *matching* input packets against a lookup

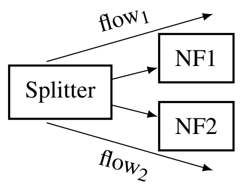


Figure 2: A sample packet processing data flow graph: a two-way splitter with two network functions (NFs). Both NFs incur one unit of execution cost per each processed batch and another one unit per each packet in the batch; the splitter incurs negligible cost. There are two service chains (or flows), one taking the upper and one taking the lower path.

table and *distributing* them to multiple next-stage processing modules may cause packet batches to fragment [20]. Worse still, fragmentation at subsequent match-tables combine multiplicatively; e.g., VRF-splitting on 16 VLANs followed by an IP lookup on 16 next-hops may easily break up a single batch of 256 packets into 256 distinct batches containing one packet each. Processing packets in the resultant small chunks takes a huge toll on the compute efficiency of the pipeline, which was designed and optimized for batch-processing in the first place [2, 11, 27]. We stress that batch-fragmentation is quite dynamic, depending on the input traffic profile, the offered load, the packet processing graph, the NFs, and flow table configuration; therefore, modeling and quantifying the resultant performance loss is challenging [22].

Trivially, *fragmented batches can be "de-fragmented" using queuing*, which delays the execution of an operation until enough packets line up at the input [1, 6, 15, 50]. This way, processing occurs over larger batches, leading to potentially significant batch-processing gains. However, queuing packets, thereby artificially slowing down a pipeline in order to speed it up, is tricky [8, 11]; a suboptimal queuing decision can easily cause delay to skyrocket. Fig. 2 shows a motivating example: if two batches containing 2 packets each enter the pipeline then unbuffered execution incurs 8 units of execution cost, as the splitter breaks up each batch into two batches containing one packet each. Placing a queue at the NF inputs, however, enables recovery of the full batches, bringing the execution cost down to 6 units (1 unit per the single batch processed and 2 units per the 2 packets in the batch) but increasing delay to 2 full turnaround times. For a k-way splitter the cost of an unbuffered execution over k batches including k packets each would be $2k^2$, which buffering would reduce to $k+k^2$; about $2\times$ batch-processing gain at the cost of $k\times$ queuing delay.

Of course, packet processing cannot be delayed for an arbitrarily long time to recover batches in full, since this may violate the service level objectives (SLOs) posed by different applications. A typical tactile internet and robot control use case requires 1–10 msec one-way, end-to-end, 99th percentile latency [28]; the 5G radio access/mobile core requires 5–10 msec; and reliable speech/video transport requires delay below 100-200 msec. At the extreme, certain industry automation, 5G inter-base-station and antenna synchronization, algorithmic stock trading, and distributed memory cache applications limit the one-way latency to 10-100 μsec [12, 25].

The key observation in this paper is that *optimal batch-scheduling in a packet processing pipeline is a fine balancing act to control queue backlogs, so that processing occurs in as large batches as possible while each flow traversing the pipeline is scheduled just fast enough to comply with the SLOs*. We present Batchy, a run-to-completion batch-scheduler framework for controlling execution in a packet-processing pipeline based on strict service-level objectives. Our contributions in Batchy are as follows:

Quantifying batch-processing gain. We observe that batch-processing efficiency varies widely across different packet-processing functions. We introduce the *Batchy profiler*, a framework for quantifying the batched service time profile for different packet-processing functions.

Analytical model. We introduce an expressive mathematical model for SLO-based batch-scheduling, so that we can reason about the performance and delay analytically and fine tune batch de-fragmentation subject to delay-SLOs. We also fix the set of basic assumptions under which the optimal schedule is well-defined (see earlier discussion in [6, 15, 50]).

Batch-processing in run-to-completion mode. Taking inspiration from Nagle's algorithm [32], we introduce a new queuing abstraction, the fractional buffer, which allows us to control queue backlogs at a fine granularity even in run-to-completion scheduling, which otherwise offers very little control over when particular network functions are executed.

Design, implementation, and evaluation of Batchy. We present a practical implementation of our batch-scheduling framework and, taking use cases from an official 5G NFV benchmark suite (L2/L3 gateway with and without ACL, NAT, VRF, a mobile gateway, and a robot-control pipeline), we demonstrate that Batchy efficiently exploits batch-processing gain consistently across a wide operational regime with small controller overhead, bringing 1.5–3$\times$ performance improvement compared to outliers and earlier work [21], while satisfying SLOs. Batchy is available for download at [4].

The rest of the paper is structured as follows. In Section 2 we introduce the Batchy profiler, Section 3 presents the idealized mathematical model and introduces fractional buffers, Section 4 discusses our implementation in detail, and Section 5 describes our experiments on real-life use cases. Finally, Section 6 discusses related work and Section 7 concludes the paper. A detailed exposition of the algorithms used in our implementation is given in the Appendix.

2 Profiling Batch-processing Gain

There are many factors contributing to the efficiency of batch-processing; next, we highlight some of the most important

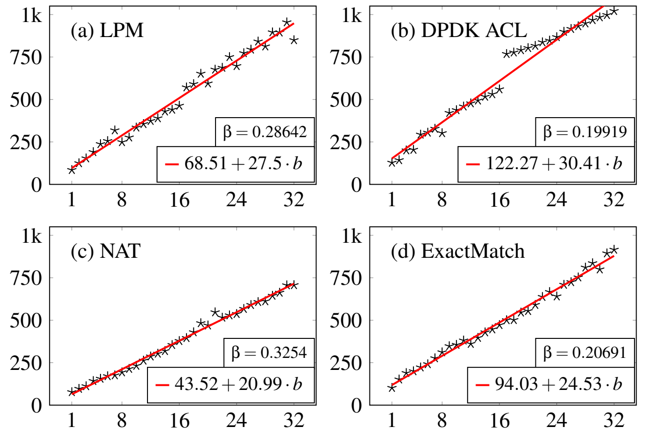


Figure 3: Service-time profile: execution time [nsec] for different modules as the function of the input batch size, averaged over 10 runs at 100,000 batches per second. The inset gives the batchiness β_v and the linear regression $T_{v,0} + T_{v,1}b_v$. Observe the effects of quad-loop/SIMD optimization for the ACL module at batch size 4, 8, and 16.

ones [26,47]. First, executing a network function on a batch incurs non-negligible computational costs independent from the number of packets in it, in terms of CPU-interrupt, scheduling, function call, I/O, memory management, and locking overhead, and *batching amortizes this fixed-cost component* over multiple packets [8, 26]. Second, executing an operation on multiple packets in one turn *improves CPU cache usage*: packets can be prefetched from main memory ahead of time and data/code locality improves as CPU caches are populated by the first packet and then further processing happens without cache misses. For example, VPP modules are written so that the entire code fits into the instruction cache, reducing icache misses [2, 27]. Third, *loop unrolling*, a compiler optimization to rewrite loops into dual- or quad-loops [26] to improve branch predictor performance and keep the CPU pipeline full, is effective only if there are multiple packets to process in one shot. Batch-processing also opens the door to exploit data-level parallelism, whereby the CPU performs the same operation on a batch of 4–32 packets in parallel for the cost of a single SIMD instruction (SSE/AVX) [16].

Intuitively, different packet-processing functions may benefit differently from batch-processing; e.g., a module processing packets in a tight loop may benefit less than a heavily SIMD-optimized one. This will then affect batch-scheduling: reconstructing batches at the input of a lightweight module might not be worth the additional queuing delay, as there is very little efficiency gain to be obtained this way.

Fig. 3 provides a *service time profile* as the execution time for some select BESS modules [14] as the function of the batch size [22]. We observe two distinct execution time components. The *per-batch cost component*, denoted by $T_{v,0}$ [sec] for a module v, characterizes the constant cost that is incurred just for calling the module on a batch, independently from the number of packets in it. The *per-packet cost component* $T_{v,1}$, [sec/pkt], on the other hand, models the execution cost of each individual packet in the batch. A linear model seems a good fit for the service time profiles: accordingly we shall use the linear regression $T_v = T_{v,0} + T_{v,1}b_v$ [sec] to describe the execution cost of a module v where b_v is the batch-size, i.e., the average number of packets in the batches received by module v. The coefficient of determination R^2 is above 96% in our tests, indicating a good fit for the linear model.

The per-batch and per-packet components determine the potential batch-processing gain on different packet processing modules. We quantify this gain with the *batchiness* measure β_v, the ratio of the effort needed to process B packets in a single batch of size B through v compared to the case when processing occurs in B distinct single-packet batches:

$$\beta_v = \frac{T_{v,0} + B * T_{v,1}}{B(T_{v,0} + T_{v,1})} \sim \frac{T_{v,1}}{T_{v,0} + T_{v,1}} \text{ for large } B \quad . \tag{1}$$

Batchiness varies between 0 and 1; small β_v indicates substantial processing gain on v and hence identifies a potential control target. The relatively small batchiness measures in Fig. 3 suggest that most real-world packet-processing functions are particularly sensitive to batch size.

Batchy contains a built-in profiler that runs a standard benchmark on the system under test at the time of initialization, collects per-batch and per-packet service-time components for common NFs, and stores the results for later use.

3 Batch-scheduling in Data Flow Graphs

Next, we present a model to describe batch-based packet processing systems. The model rests on a set of simplifying assumptions, which prove crucial to reason about such systems using formal arguments. We adopt the terminology and definitions from BESS, but we note that the model is general enough to apply to most popular data flow graph packet-processing engines, like VPP [47], Click/FastClick [3, 31], NetBricks [36], or plain DPDK [16]; match-action pipelines like Open vSwitch [38] or ESwitch [30]; or to data flow processing frameworks beyond the networking context like TensorFlow [10] or GStreamer [43].

3.1 System model

Data flow graph. We model the pipeline as a directed graph $\mathcal{G} = (V, E)$, with modules $v \in V$ and directed links $(u, v) \in E$ representing the connections between modules. A *module* v is a combination of a (FIFO) *ingress queue* and a *network function* at the egress connected back-to-back (see Fig. 4). *Input gates* (or ingates) are represented as in-arcs $(u, v) \in E$: $u \in V$ and *output gates* (or outgates) as out-arcs $(v, u) \in E$:

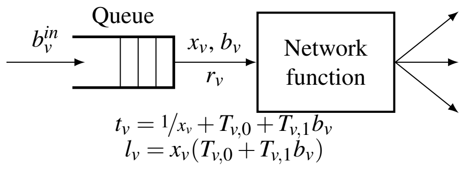

Queue
b_v^{in}
x_v, b_v
r_v
Network function
$t_v = 1/x_v + T_{v,0} + T_{v,1} b_v$
$l_v = x_v(T_{v,0} + T_{v,1} b_v)$

Figure 4: A Batchy module.

$u \in V$. A batch sent to an outgate (v,u) of v will appear at the corresponding ingate of u at the next execution of u. Modules never drop packets; we assume that whenever an ACL module or a rate-limiter would drop a packet it will rather send it to a dedicated "drop" gate, so that we can account for lost packets. A normal queue is a module with an empty network function.

Batch processing. Packets are injected into the ingress, transmitted from the egress, and processed from outgates to ingates along data flow graph arcs, in batches [2, 14, 16, 27, 30]. We denote the maximum batch size by B, a system-wide parameter. For the Linux kernel and DPDK $B = 32$ or $B = 64$ are usual settings, VPP sets the batch size to 256 packets by default [27], while GPU/NIC offload often works with $B = 1024$ or even larger to maximize I/O efficiency [40, 50].

Splitters/mergers. Any module may have multiple ingates (merger) and/or multiple outgates (splitter), or may have no ingate or outgate at all. An IP Lookup module would distribute packets to several downstream branches, each performing group processing for a different next-hop (splitter); a NAT module may multiplex traffic from multiple ingates (merger); and an IP Checksum module would apply to a single datapath flow (single-ingate–single-outgate). Certain modules are represented without ingates, such as a NIC receive queue; we call these *ingress modules* S. Similarly, a module with no outgates (e.g., a transmit queue) is an *egress module*.

Compute resources. A *worker* is an abstraction for a CPU core, where each worker $w \in \mathcal{W}$ is modeled as a connected subgraph $\mathcal{G}_w = (V_w, E_w)$ of $\mathcal{G}$ with strictly one ingress module $S_w = \{s_w\}$ executing on the same CPU. We assume that when a data flow graph has multiple ingress modules then each ingress is assigned to a separate worker, with packets passing between workers over double-ended queues. A typical setup is to dedicate a worker to each receive queue of each NIC and then duplicate the entire data flow graph for each worker. Each worker may run a separate explicit scheduler to distribute CPU time across the modules in the worker graph, or it may rely on run-to-completion; see Appendix A for an overview.

Flows. A flow $f = (p_f, R_f, D_f), f \in \mathcal{F}$ is our abstraction for a service chain, where p_f is a path through $\mathcal{G}$ from the flow's ingress module to the egress module, R_f denotes the offered packet rate at the worker ingress, and D_f is the delay-SLO, the maximum permitted latency for any packet of f to reach the egress. What constitutes a flow, however, will be use-case specific: in an L3 router a flow is comprised of all traffic destined to a single next-hop or port; in a mobile gateway a flow is a complex combination of a user selector and a bearer selector; in a programmable software switch flows are completely configuration-dependent and dynamic. Correspondingly, flow-dissection in a low-level packet processing engine cannot rely on the RSS/RPS function supplied by the NIC, which is confined to VLANs and the IP 5-tuple [6, 19]. Rather, in our framework flow dispatching occurs *intrinsically* as part of the data flow graph; accordingly, we presume that match-tables (splitters) are set up correctly to ensure that the packets of each flow f will traverse the data flow graph along the path p_f associated with f.

3.2 System variables

We argue that at multiple gigabits per second it is overkill to model the pipeline at the granularity of individual packets [22]. Instead, in our model variables are continuous and differentiable, describing system *statistics* over a longer period of time that we call the *control period*. This is analogous to the use of standard (continuous) network flow theory to model packet routing and rate control problems. We use the following variables to describe the state of the data flow graph in a given control period (dimensions indicated in brackets).

Batch rate x_v [$1/s$]: the number of batches per second entering the network function in module v (see again Fig. 4).

Batch size b_v [pkt]: the average number of packets per batch at the input of the network function in module v, where $b_v \in [1, B]$ and, recall, B is the maximum allowed batch size.

Packet rate r_v [pkt/s]: the number of packets per second traversing module v: $r_v = x_v b_v$.

Maximum delay t_v [sec]: delay contribution of module v to the total delay of packets traversing it. We model t_v as

$$t_v = t_{v,\text{queue}} + t_{v,\text{svc}} = 1/x_v + (T_{v,0} + T_{v,1} b_v) \quad , \tag{2}$$

where $t_{v,\text{queue}} = 1/x_v$ is the queuing delay by Little's law and $t_{v,\text{svc}} = T_{v,0} + T_{v,1} b_v$ is the service time profile (see Section 2).

System load l_v (dimensionless): the network function in module v with service time $t_{v,\text{svc}}$ executed x_v times per second incurs $l_v = x_v t_{v,\text{svc}} = x_v(T_{v,0} + T_{v,1} b_v)$ system load in the worker.

Turnaround-time T_0 [sec]: the maximum CPU time the system may spend pushing a single batch through the pipeline. The turnaround time typically varies with the type and number of packets in each batch, the queue backlogs, etc.; correspondingly, we usually consider the time to execute *all* modules on maximum sized batches as an upper bound:

$$T_0 \le \sum_{v \in V} (T_{v,0} + T_{v,1} B) \quad . \tag{3}$$

3.3 Assumptions

Our aim is to define the simplest possible batch-processing model that still allows us to reason about flows' packet rate and maximum delay, and modules' batch-efficiency. The below assumptions will help to keep the model at the minimum; these assumptions will be relaxed later in Section 4.

Feasibility. We assume that the pipeline runs on a single worker and this worker has enough capacity to meet the delay-SLOs. In the next section, we show a heuristic method to decompose a data flow graph to multiple workers in order to address SLO violations stemming from inadequate resources.

Buffered modules. We assume that all modules contain an ingress queue and all queues in the pipeline can hold up to at most B packets at any point in time. In the next section, we show how to eliminate useless queues in order to remove the corresponding latency and processing overhead.

Static flow rate. All flows are considered constant-bit-rate (CBR) during the control period (usually in the millisecond time frame). This assumption will be critical for the polynomial tractability of the model. Later on, we relax this assumption by incorporating the model into a receding-horizon optimal control framework.

3.4 Optimal explicit batch-schedule

Workers typically run an *explicit scheduler* to distribute CPU time across the modules in the worker graph. Common examples include Weighted Fair Queueing (WFQ) and Completely Fair Scheduling (CFS); here, the user assigns integer weights to modules and the scheduler ensures that runtime resource allocation will be proportional to modules' weight [46]. For simplicity, we consider an *idealized* WFQ/CFS scheduler instead, where execution order is defined in terms per-module *rates* and not weights; rates will be converted to weights later.

The idealized scheduler runs each module precisely at the requested rate. When scheduled, the modules' network function dequeues at most a single batch worth of packets from the ingress queue, executes the requested operation on all packets of the batch, forms new sub-batches from processed packets and places these to the appropriate outgates.

In this setting, we seek for a set of rates x_v at which each module $v \in V$ needs to be executed in order to satisfy the SLOs. If multiple such rate allocations exist, then we aim to choose the one that minimizes the overall system load.

Recall, executing v exactly x_v times per second presents $l_v = x_v t_{v,\text{svc}} = x_v(T_{v,0} + T_{v,1}b_v)$ load to the system. The objective function, correspondingly, is to find rates x_v that minimize the total system load $\sum_{v \in V} l_v$, taken across all modules:

$$\min \sum_{v \in V} x_v(T_{v,0} + T_{v,1}b_v) \ . \tag{4}$$

Once scheduled, module v will process at most $b_v \in [1,B]$ packets through the network function, contributing $t_{v,\text{svc}} = T_{v,0} + T_{v,1}b_v$ delay to the total latency of each flow traversing it. In order to comply with the delay-SLOs, for each flow f it must hold that the total time spent by any packet in the worker ingress queue, plus the time needed to send a packet through the flow's path p_f, must not exceed the delay requirement D_f for f. Using that the ingress queue of size B may develop a backlog for only at most one turnaround time T_0 (recall, we assume there is a single worker and each queue holds at most B packets), and also using (2), we get the following *delay-SLO constraint*:

$$t_f = T_0 + \sum_{v \in p_f} \left({}^{1}\!/_{x_v} + T_{v,0} + T_{v,1}b_v\right) \le D_f \qquad \forall f \in \mathcal{F} \ . \tag{5}$$

Each module $v \in V$ must be scheduled frequently enough so that it can handle the *total offered packet rate* $R_v = \sum_{f: v \in p_f} R_f$, i,e., the sum of the requested rate R_f of each flow f traversing v (recall, we assume flow rates R_f are constant). This results the following *rate constraint*:

$$r_v = x_v b_v = \sum_{f: v \in p_f} R_f = R_v \qquad \forall v \in V \ . \tag{6}$$

Finally, the backlog b_v at any of the ingress queues across the pipeline can never exceed the queue size B and, of course, all system variables must be non-negative:

$$1 \le b_v \le B, \ x_v \ge 0 \qquad \forall v \in V \ . \tag{7}$$

Together, (4)–(7) defines an optimization problem which provides the required static scheduling rate x_v and batch size b_v for each module v in order to satisfy the SLOs while maximizing the batch-processing gain. This of course needs the turnaround time T_0; one may use the approximation (3) to get a conservative estimate. Then, substituting $b_v = \sum_{f: v \in p_f} {}^{R_f}\!/_{x_v} = {}^{R_v}\!/_{x_v}$ using (6), we get the following system, now with only the batch-scheduling rates x_v as variables:

$$\min \sum_{v \in V} x_v\left(T_{v,0} + T_{v,1}\frac{R_v}{x_v}\right) \tag{8}$$

$$t_f = T_0 + \sum_{v \in p_f} \left(\frac{1}{x_v} + T_{v,0} + T_{v,1}\frac{R_v}{x_v}\right) \le D_f \quad \forall f \in \mathcal{F} \tag{9}$$

$${}^{R_v}\!/_{B} \le x_v \le R_v \qquad \forall v \in V \tag{10}$$

Since the constraints and the objective function are convex, we conclude that (8)–(10) *is polynomially tractable and the optimal explicit batch-schedule is unique* [5]. Then, setting the scheduling weights proportionally to rates x_v results in the optimal batch-schedule on a WFQ/CFS scheduler [46].

3.5 Run-to-completion execution

WFQ/CFS schedulers offer a plausible way to control batch de-fragmentation via the per-module weights. At the same time, often additional tweaking is required to avoid head-of-line blocking and late drops along flow paths [21], and even running the scheduler itself may incur non-trivial runtime overhead. *Run-to-completion execution*, on the other hand, eliminates the explicit scheduler all together, by tracing the entire input batch though the data flow graph in one shot without the risk of head-of-line blocking and internal packet drops [6,14]. Our second batch-scheduler will therefore adopt run-to-completion execution.

The idea in run-to-completion scheduling is elegantly simple. The worker checks the input queue in a tight loop and, whenever the queue is not empty, it reads a single batch and injects it into pipeline at the ingress module. On execution, each module will process a single batch, place the resulting packets at the outgates potentially breaking the input batch into multiple smaller output batches, and then recursively schedule the downstream modules in order to consume the sub-batches from the outgates. This way, the input batch proceeds through the entire pipeline in a single shot until the last packet of the batch completes execution, at which point the worker returns to draining the ingress queue. Since upstream modules will *automatically* schedule a downstream module whenever there is a packet waiting to be processed, run-to-completion execution does not permit us to control when individual modules are to be executed. This makes it difficult to enforce SLOs, especially rate-type SLOs, and to delay module execution to de-fragment batches.

Below, we introduce a new queuing abstraction, the *fractional buffer*, which nevertheless lets us exert fine-grained control over modules' input batch size. The fractional buffer is similar to Nagle's algorithm [32], originally conceived to improve the efficiency of TCP/IP networks by squashing multiple small messages into a single packet. The backlog is controlled so as to keep end-to-end delay reasonable. Indeed, Nagle's algorithm exploits the same batch-efficiency gain over the network as we intend to exploit in the context of compute-batching, motivating our choice to apply it whenever there is sufficient latency slack available.

A fractional buffer maintains an internal FIFO queue and exposes a single parameter to the control plane called the *trigger* b, which enables tight control of the queue backlog and thereby the delay. The buffer will enqueue packets and suppress execution of downstream modules until the backlog reaches b, at which point a packet batch of size b is consumed from the queue, processed in a single burst through the succeeding module, and execution of downstream modules is resumed. Detailed pseudocode is given in Appendix C.

We intentionally define the trigger in the batch-size domain and not as a perhaps more intuitive timeout [32], since timeouts would re-introduce an explicit scheduler into the otherwise "schedulerless" design. Similarly, we could in theory let the buffer to emit a batch larger than b whenever enough packets are available; we intentionally restrict the output batch to size b so as to tightly control downstream batch size.

What remains is to rewrite the optimization model (8)–(10) from explicit module execution rates x_v to fractional buffer triggers. Interestingly, *jumping from rate-based scheduling to the run-to-completion model is as easy as substituting variables*: if we replace the ingress queue with a fractional buffer with trigger b_v in each module v, then the subsequent network function will experience a batch rate of $x_v = {R_v}/{b_v}$ at batch size b_v. Substituting this into the optimization problem (4)–(7) yields the *optimal batch-schedule for the run-to-completion model* with variables $b_v : v \in V$:

$$\min \sum_{v \in V} \frac{R_v}{b_v}(T_{v,0} + T_{v,1} b_v) \tag{11}$$

$$t_f = T_0 + \sum_{v \in p_f} \left(\frac{b_v}{R_v} + T_{v,0} + T_{v,1} b_v \right) \le D_f \quad \forall f \in \mathcal{F} \tag{12}$$

$$1 \le b_v \le B \quad \forall v \in V \tag{13}$$

Again, the optimization problem is convex and the optimal setting for the fractional buffer triggers is unique. In addition, the feasible region is linear, which makes the problem tractable for even the simplest of convex solvers. Since the substitution $x_v = {R_v}/{b_v}$ can always be done, we find that for *any* feasible batch-rate $x_v : v \in V$ in the explicit scheduling problem (8)–(10) there is an equivalent set of fractional buffer triggers $b_v : v \in V$ in the run-to-completion problem (11)–(13) and *vice versa*; put it another way, *any system state (collection of flow rates and delays) feasible under the explicit scheduling model is also attainable with a sufficient run-to-completion schedule*. To the best of our knowledge, this is the first time that an equivalence between the two crucial data flow-graph scheduling models is shown. We note however that the assumptions in Section 3.3 are critical for the equivalence to hold. For instance, explicit scheduling allows for a "lossy" schedule whereas a run-to-completion schedule is lossless by nature; under the "feasibility" assumption, however, there is no packet loss and hence the equivalence holds.

4 Implementation

We now describe the design and implementation of Batchy, our batch-scheduler for data flow graph packet-processing engines. The implementation follows the model introduced above, extended with a couple of heuristics to address practical limitations. We highlight only the main ideas of the implementation below; a detailed description of the heuristics with complete pseudocode can be found in the Appendix.

Design. To exploit the advantages of the simple architecture and zero scheduling overhead, in this paper we concentrate on the run-to-completion model (11)–(13) exclusively and we leave the implementation of the explicit scheduling model (8)–(10) for further study. This means, however, that currently we can enforce only delay-type SLOs using Batchy. In our design, the control plane constantly monitors the data plane and periodically intervenes to improve batch-efficiency and satisfy SLOs. Compared to a typical scheduler, Batchy interacts with the data plane at a coarser grain: instead of operating at the granularity of individual batches, it controls the pipeline by periodically adjusting the fractional buffer triggers and then it relies entirely on run-to-completion scheduling to handle the fine details of module execution.

Receding-horizon control. A naive approach to implement the control plane would be to repeatedly solve the convex program (11)–(13) and apply the resulting optimal fractional-buffer triggers to the data plane. Nevertheless, by the time the convex solver finishes computing the optimal schedule the system may have diverged substantially from the initial state with respect to which the solution was obtained. To tackle this difficulty, we chose a *one-step receding-horizon control* framework to implement Batchy. Here, in each control period the optimization problem (11)–(13) is bootstrapped with the current system state and fed into a convex solver, which is then is stopped *after the first iteration*. This results a coarse-grain control action, which is then immediately applied to the data plane. After the control period has passed, the system is re-initialized from the current state and a control is calculated with respect to this new state. This way, the controller rapidly drives the system towards improved states and eventually reaches optimality in steady state, automatically adapting to changes in the input parameters and robustly accounting for inaccuracies and failed model assumptions without having to wait for the convex solver to fully converge in each iteration.

Main control loop. Upon initialization, Batchy reads the data flow graph, the flows with the SLOs, and per-module service time profiles from the running system. During runtime, it measures in each control period the execution rate $\tilde{x}_v$, the packet rate $\tilde{r}_v$, and the mean batch size $\tilde{b}_v^{in}$ at the *input* of the ingress queue for each module $v \in V$, plus the 95th percentile packet delay $\tilde{t}_f$ measured at the egress of each flow $f \in \mathcal{F}$. (The overbar tilde notation is to distinguish *measured* parameters.) The statistics and the control period are configurable; e.g., Batchy can be easily re-configured to control for the 99th percentile or the mean delay. Due to its relative implementation simplicity and quick initial convergence, we use the gradient projection algorithm [5] to compute the control but in each run we execute only a *single* iteration. The algorithm will adjust triggers so as to obtain the largest relative gain in total system load and, whenever this would lead to an SLO violation, cut back the triggers just enough to avoid infeasibility.

Insert/short-circuit buffers. An unnecessary buffer on the packet-processing fast path introduces considerable delay and incurs nontrivial runtime overhead. In this context, a buffer is "unnecessary" if it already receives large enough batches at the input (Batchy detects such cases by testing for $\tilde{b}_v^{in} \geq 0.7B$); if it would further fragment batches instead of reconstructing them ($b_v \leq \tilde{b}_v^{in}$); or if just introducing the buffer already violates the delay-SLO ($1/x_v > D_f$ for some $v \in p_f$). If one of these conditions hold for a module v, Batchy immediately short-circuits the buffer in v by setting the trigger to $b_v = 0$: the next-time module v is executed the ingress queue is flushed and subsequent input batches are immediately fed into the network function without buffering. Similar heuristics allow Batchy to inject buffers into the running system: at initialization we short-circuit all buffers ("null-control", see below) and, during runtime, we install a buffer whenever *all* flows traversing a module provide sufficient delay-budget.

Recovering from infeasibility. The projected gradient controller cannot by itself recover from an infeasible (SLO-violating) state, which may occur due to packet rate fluctuation or an overly aggressive control action. A flow f is in SLO-violation if $\tilde{t}_f \geq (1-\delta)D_f$ where δ is a configurable parameter that allows to trade off SLO-compliance for batch-efficiency. Below, we use the setting $\delta = 0.05$, which yields a rather aggressive control that strives to maximize batch size with a tendency to introduce relatively frequent, but small, delay violations. Whenever a flow $f \in \mathcal{F}$ is in SLO violation and there is a module v in the path of f set to a non-zero trigger ($b_v > \tilde{b}_v^{in}$), we attempt to reduce b_v by $\left\lceil \frac{D_f - t_f}{\partial t_f / \partial b_v} \right\rceil$. If possible, this would cause f to immediately recover from the SLO-violation. Otherwise, it is possible that the invariant $b_v \geq \tilde{b}_v^{in}$ may no longer hold; then we repeat this step at as many modules along p_f as necessary and, if the flow is still in SLO-violation, we short-circuit all modules in p_f.

Pipeline decomposition. Batchy contains a pipeline controller responsible for migrating flows between workers to enforce otherwise unenforceable delay-SLOs. Consider the running example in Fig. 2, assume a single worker, let the processing cost of NF1 be 1 unit and that of NF2 be 10 units, and let the delay-SLO for the first flow be 2 units. This pipeline is inherently in delay-SLO violation: in the worst case a packet may need to spend 10 time units in the ingress queue until NF2 finishes execution, significantly violating the delay-SLO for the first flow (2 units). This inherent SLO violation will persist as long as NF1 and NF2 share a single worker. Batchy uses the analytical model to detect such cases: a worker w is in inherent delay-SLO violation if there is a flow $f \in \mathcal{F}$ for which $\sum_{v \in V_w}(T_{v,0} + T_{v,1}B) \geq D_f$ holds, using the conservative estimate (3). Then Batchy starts a flow migration process: first it packs flows back to the original worker as long as the above condition is satisfied and then the rest of the flows are moved to a new worker. This is accomplished by decomposing the data flow graph into multiple disjunct connected worker subgraphs. Note that flows are visited in the ascending order of the delay-SLO, thus flows with restrictive delay requirements will stay at the original worker with a high probability, exempt from cross-core processing delays [19].

Implementation. We implemented Batchy on top of BESS [14] in roughly 6,000 lines of Python/BESS code. (Batchy is available at [4].) BESS is a high-performance packet processing engine providing a programming model and interface similar to Click [31]. BESS proved an ideal prototyping platform: it has a clean architecture, is fast [24], provides an efficient scheduler, exposes the right abstractions and offers a flexible plugin infrastructure to implement the missing ones. The distribution contains two built-in controllers. The *on-off controller* ("Batchy/on-off") is designed for the case when fractional buffers are not available in the data plane. This controller alters between two extremes (bang-bang control): at each module v, depending on the delay budget it either disables buffering completely ($b_v = 0$) or switches to full-batch buffering ($b_v = B$), using the above buffer-insertion/deletion and feasibility-recovery heuristics. On top of this, the *full-fledged Batchy controller* ("Batchy/full") adds fractional buffers and fine-grained batch-size control using the projected gradient method.

5 Evaluation

Batchy is a control framework for packet-processing engines, which, depending on flows' offered packet rate and delay-SLOs, searches for a schedule that balances between batch-processing efficiency and packet delay. In this context the following questions naturally arise: *(i)* how much do batches fragment in a typical use case (if at all) and how much efficiency is there to gain by reconstructing these? how precisely does Batchy enforce SLOs?; *(ii)* which is the optimal operational regime for Batchy and what is the cost we pay?; *(iii)* does Batchy react quickly to changes in critical system parameters?; and finally *(iv)* can Batchy recover from inherent SLO-violations? Below we seek to answer these questions.

Evaluation setup. To understand the performance and latency-related impacts of batch control, we implemented two baseline controllers alongside the basic Batchy controllers (Batchy/on-off and Batchy/full): the *null-controller* performs no batch de-fragmentation at all ($b_v = 0 : v \in V$), while the *max-controller* reconstructs batches in full at the input of all modules ($b_v = B : v \in V$). Both baseline controllers ignore SLOs all together. The difference between performance and delay with the null- and max-controllers will represent the maximum attainable efficiency improvement batching may yield, and the price we pay in terms of delay. We also compared Batchy to NFVnice, a scheduling framework originally defined for the NFV context [21]. NFVnice is implemented in a low-performance container-based framework; to improve performance and to compare it head-to-head, we re-implemented its core functionality within BESS. Our implementation uses WFQ to schedule modules with equal weights and enables backpressure. All controllers run as a separate Python process, asserting *real-time control* over the BESS data plane via an asynchronous gRPC channel.

The evaluations are performed on 5 representative use cases taken from an official industry 5G benchmark suite [24]. The L2/L3(n) pipeline implements a basic IP router, with L2 lookup, L3 longest-prefix matching, and group processing for n next-hops; the GW(n) use case extends this pipeline into a full-fledged gateway with NAT and ACL processing for n next-hop groups; and the VRF(m,n) pipeline implements m virtual GW(n) instances preceded by an ingress VLAN splitter (see Fig. 5). The MGW(m,n) pipeline is a full 4G mobile gateway data plane with m users and n bearers per

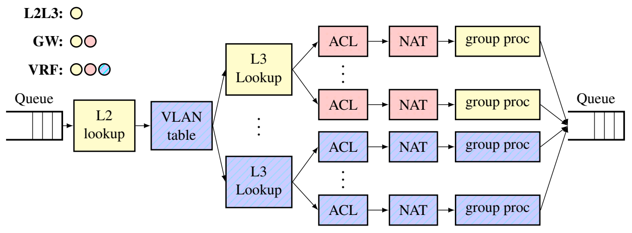


Figure 5: The VRF pipeline. The GW pipeline is identical to the VRF pipeline with only a single VRF instance, and the L2/L3 pipeline is a GW pipeline without a NAT and ACL on the next-hop branches.

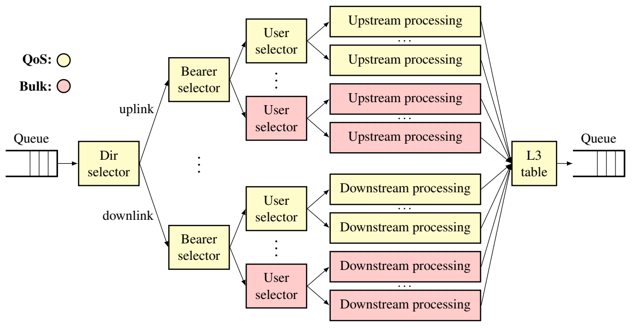


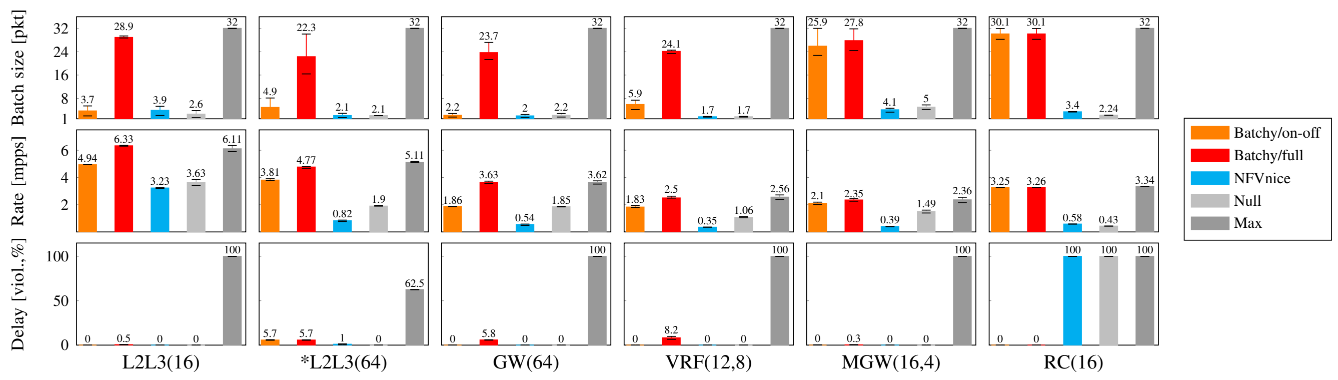
Figure 6: The mobile gateway (MGW) pipeline.

user, with complete uplink and downlink service chains (see Fig. 6). Finally, the RC(n) pipeline models a 5G robot control use case: RC(n) corresponds to the running example in Fig. 2 with n branches, with the upper branch representing an ultra-delay-sensitive industry automation service chain and the rest of the branches carrying bulk traffic. We obtained test cases of configurable complexity by varying the parameters m and n and installing a flow over each branch of the resultant pipelines; e.g., in the VRF(2,4) test we have a separate flow for each VRF and each next-hop, which gives 8 flows in total.

Each pipeline comes with a traffic source that generates synthetic test traffic for the evaluations. For the L2/L3 pipeline, we repeated the tests with a real traffic trace taken from a CAIDA data set [9], containing 1.85 million individual transport sessions of size ranging from 64 bytes to 0.8 Gbytes (18 Kbyte mean) and maximum duration of 1 min (5.4 sec mean); the results are marked with the label *L2L3. Unless otherwise noted, the pipelines run on a single worker (single CPU core), with the traffic generator provisioned at another worker. The maximum batch size is 32, the control period is 100 msec, and results are averaged over 3 consecutive runs.

Each evaluation runs on a server equipped with an Intel Xeon E5-2620 v3 CPU (12 cores total, 6 isolated, power-saving disabled, single socket) with 64GB memory (with 12 $\times$ 1GB allocated as hugepages), installed with the Debian/GNU operating system, Linux kernel v4.19, a forked version of BESS v0.4.0-57-g43bebd3, and DPDK v17.11.

Batch-scheduling constant bit-rate flows. In this test round, we benchmark the static performance of the Batchy con-



Common parameters. test time: 200 periods, warmup: 100 periods, control period: 0.1sec, $\delta = 0.05$, SLO-control: 95th percentile delay, burstiness: 1. Measure params (bucket/max): 5μsec/250msec. L2L3: FIB: 500 entries. CAIDA trace: `equinix-nyc.dirA.20190117-130600.UTC.anon`. GW: ACL: 100 entries, NAT: static NAT. MGW: FIB: 5k entries, 4 users on bearer0, upstream/downstream processing $T_{0,v} = 2000, T_{1,v} = 100$.

Figure 7: Static evaluation results (mean, 1st and 3rd quartile) with the null-controller, max-controller, NFVnice, Batchy/on-off and Batchy/full on different pipelines: average batch-size, cumulative packet rate, and delay-SLO statistics as the percentage of control periods when an SLO-violation was detected for at least one flow.

trollers against the baselines and NFVnice over 6 representative 5G NFV configurations. The null-controller, the max-controller, and NFVnice do not consider the delay-SLO, while for Batchy/on-off and Batchy/full we set the delay-SLO to 80% of the average delay measured with the max-controller; this setting leaves comfortable room to perform batch de-fragmentation but sets a firm upper bound on triggers. (Without an SLO, Batchy controllers degrade into a max-controller.)

After a warmup (100 control periods) we ran the pipeline for 100 control periods and we monitored the batch size statistics averaged across modules, the cumulative packet rate summed over all flows, and the delay-SLO statistics as the percentage of control periods when an SLO violation occurs for *at least* one flow. Fig. 7 highlights the results for 6 select configurations and Appendix B details the full result set. Our observations are as follows.

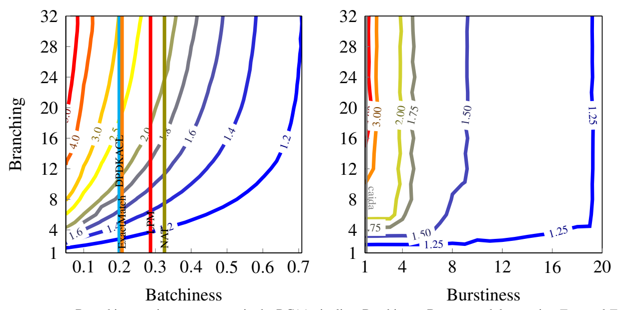
First, *the full-fledged Batchy controller (Batchy/full) can successfully reconstruct batches* at the input of network functions, achieving 70–80% of the average batch size of the max-controller in essentially all use cases (same proportion as the delay-SLO constraints). *Batch-fragmentation gets worse as the number of branches increases* across which batches may be split (the *branching* factor), to the point that when there are 16–64 pathways across the data flow graph the null-controller works with just 2–3 packets per batch. The simplified controller (Batchy/on-off) produces mixed results: whenever there is enough delay-budget to insert full buffers it attains similar de-fragmentation as Batchy/full (MGW(16,4), RC(16)), while in other cases it degrades into null-control (GW(64)).

Second, *batch de-fragmentation clearly transforms into considerable efficiency improvement*. Batchy/full exhibits 1.5–2.5× performance compared to the case when we do no batch de-fragmentation at all (null-control), and Batchy/on-off shows similar, although smaller, improvements. In the robot-control use case we see 7.5× throughput margin. This experiment demonstrates the benefits of selective per-module batch-control: there is only one highly delay-sensitive flow but this alone rules out any attempt to apply batching *globally* (even at the I/O); Batchy can, however, identify this chain and short-circuit all buffers along just this chain while it can still buffer the remaining bulk flows, yielding a dramatic performance boost. Despite the firm upper bound on the delay, and on the maximum attainable batch size, Batchy performs very close to the max-controller and in some tests even outperforms it (e.g., for L2L3(16)). This is because the max-controller buffers all modules unconditionally while Batchy carefully removes unnecessary buffers, and this helps in terms of valuable CPU cycles saved. The results are consistently reproduced over both the synthetic and the real traffic traces.

Third, despite the rather aggressive controller settings ($\delta = 0.05$, see the previous Section), *Batchy controllers violate the delay-SLO at most* 9% *of time* for at least one out of the possibly 64 flows, and even in these cases the relative delay violation is always below 1–2% (not shown in the figures). We believe that this is a price worth paying for the efficiency gain; manually repeating a failed test with a less aggressive control ($\delta = 0.2$) eliminated delay-SLO violations all together, at the cost of somewhat lower throughput.

Finally, we see that the Batchy/on-off controller is already useful in its own right in that it produces substantial batch-performance boost in certain configurations, but it hardly improves on null-control in others. It seems that discrete on-off control is too coarse-grained to exploit the full potential of batch de-fragmentation; to get full advantage we need finer-grained control over batch sizes, and the ensuing delay, using fractional buffers and the Batchy/full controller.

Optimal operational regime and runtime overhead. Next, we attempt to obtain a general understanding of the efficiency improvements attainable with Batchy, and the cost we pay in



Branching: setting parameter n in the RC(n) pipeline. Batchiness: Bypass module, varying $T_{0,v}$ and $T_{1,v}$.
Burstiness: varying mean burst size. Fixed parameters: first plot: burstiness=1; second plot: $\beta = 1/11$ for all modules.

Figure 8: Batchiness and burstiness vs. branching: Batchy/full packet rate normalized to the maximally fragmented case.

terms of controller overhead. For this, we first define a set of meta-parameters that abstract away the most important factors that shape the efficiency and overhead of Batchy, and then we conduct extensive evaluations in the configuration space of these meta-parameters.

The meta-parameters are as follows. Easily, it is the complexity of the underlying data flow graph that fundamentally determines Batchy's performance. We abstract pipeline complexity using the *branching* meta-parameter, which represents the number of distinct control-flow paths through the data flow graph; the higher the branching the more batches may break up inside the pipeline and the larger the potential batch de-fragmentation gain. Second, *batchiness*, as introduced in Section 2, determines each module's sensitivity to batch size; small batchiness usually indicates huge potential de-fragmentation gain. Finally, the specifics of the input traffic pattern is captured using the *burstiness* meta-parameter, which measures the average size of back-to-back packet bursts (or flowlets [37]) at the ingress of the pipeline. Indeed, burstiness critically limits batch-efficiency gains: as packet bursts tend to follow the same path via the graph they are less prone to fragmentation, suggesting that the performance margin of de-fragmentation may disappear over highly bursty traffic.

To understand how these factors shape the efficiency of Batchy, Fig. 8 shows two contour plots; the first one characterizes the speedup with Batchy/full compared to null-control in the branching–batchiness domain and the second one measures branching against burstiness. The plots clearly outline the optimal operational regime for Batchy: *as the number of branches grows beyond* 4–8 *and batchiness remains under* 0.5 *we see* 1.5–4× *speedup, with diminishing returns as the mean burst size grows beyond* 10. These gains persist with realistic exogenous parameters; batchiness for real modules is between 0.2–0.3 (see Fig. 3) and the CAIDA trace burstiness is only 1.13 (see the vertical indicators in the plots). But even for very bursty traffic and/or poor batch sensitivity, Batchy consistently brings over 1.2× improvement and never worsens performance: for workloads that do not benefit from batch de-fragmentation Batchy rapidly removes useless buffers and falls back to default, unbuffered forwarding.

Branching (n)	Response time	Stats	Gradient	Control
1	2.4 msec	66%	31%	3%
2	3.5 msec	61%	37%	2%
4	6.9 msec	65%	32%	3%
8	11.6 msec	65%	32%	3%
16	21.9 msec	66%	30%	4%
32	34.8 msec	68%	28%	4%
64	89.1 msec	72%	22%	6%

Table 1: Batchy/full runtime overhead on increasingly more complex RC(n) pipelines: branching, total controller response time, and contribution of each phase during the control, i.e., monitoring (Stats), gradient control (Gradient), and applying the control (Control).

Table 1 summarizes the controller runtime overhead in terms of the "pipeline complexity" meta-parameter (branching). The profiling results indicate that *the performance gains come at a modest resource footprint*: depending on pipeline complexity the controller response time varies between 2–90 milliseconds, with roughly two thirds of the execution time consumed by marshaling the statistics out from the data-plane and applying the new triggers back via gRPC, and only about one third taken by running the gradient controller itself.

System dynamics under changing delay-SLOs. We found the projected gradient controller to be very fast in the static tests: whenever the system is in steady state (offered load and delay-SLOs constant), Batchy usually reaches an optimal KKT point [5] in about 5–10 control periods. In the tests below, we evaluated Batchy under widely fluctuating system load to get a better understanding of the control dynamics.

First, we study how Batchy reacts to changing delay-SLOs (see the `conf/l2l3_vd.batchy` config file in the Batchy source distribution [4]). The results are in Fig. 9a. In this experiment, we set up the VRF(4, 8) pipeline and vary the delay-SLO between 60 μsec and 300 μsec in 6 steps; see the blue dotted "square wave" in Fig. 9a/Delay panel. The SLOs were set so that we test abrupt upwards ("rising edge") and downwards ("falling edge") delay-SLO changes as well. The figure shows for each control period the value of the trigger at the ACL module of the top branch (first VRF, first next-hop), the total delay and the delay-SLO for the flow provisioned at the top branch, and the normalized cumulative packet rate with the null-controller, the max-controller, and Batchy/full.

The results suggest that *Batchy promptly reacts to "bad news"* (SLO-reduction, falling edge, Fig. 9a/Delay) and instantaneously reduces fractional buffer triggers (Fig. 9a/Control), or even completely short-circuits buffers to recover from SLO-violations, whereas it is much more *careful to react to "good news"* (increasing SLO, rising edge). Overall, the packet delay closely tracks the SLO dynamics. Meanwhile, whenever there is room to perform batch de-fragmentation Batchy rapidly reaches the efficiency of the max-controller, *delivering* 2–3× *the total throughput of the null-controller*.

System dynamics with variable bitrate traffic. Next, we

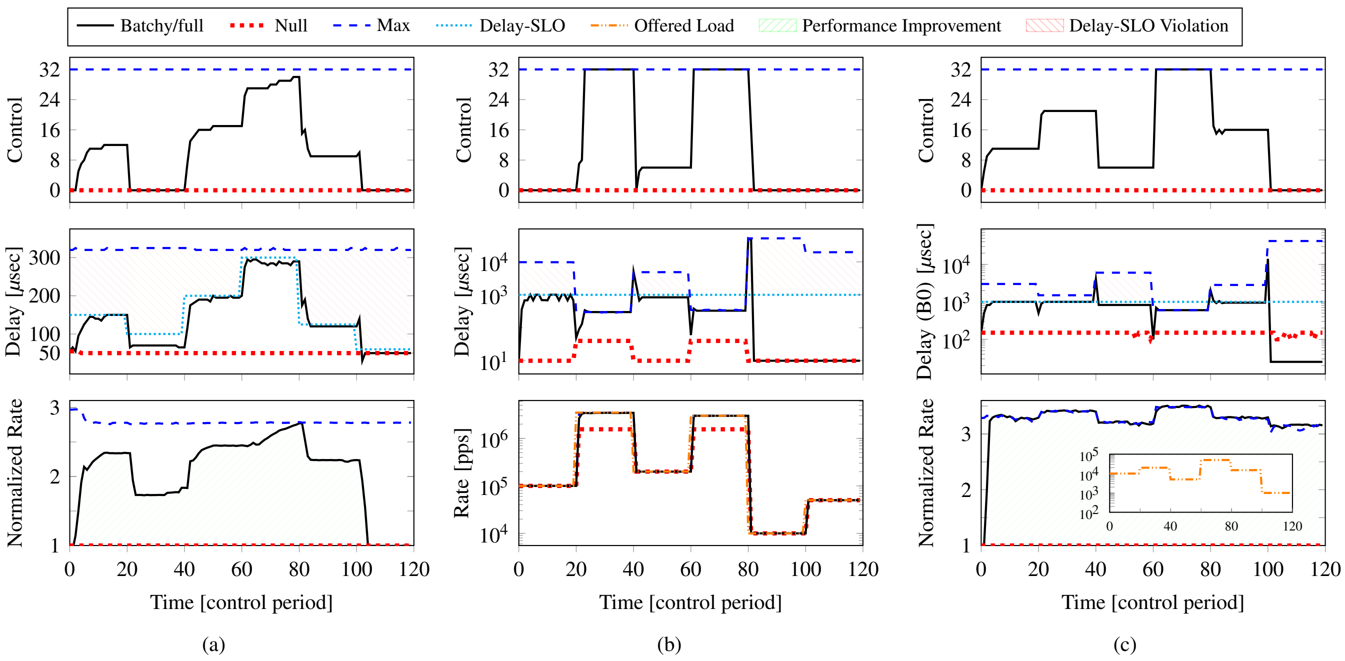


Figure 9: System dynamics with changing SLOs and variable bit-rate traffic: (a) control and delay for the first flow, and cumulative packet rate in the VRF(4,8) pipeline when the delay-SLO changes in the range 60–300 μsec; (b) control and delay for the first flow, and cumulative packet rate in the VRF(4,8) pipeline with the delay-SLO of all flows fixed at 1 msec and varying the total offered load between 50 kpps to 3 mpps; and (c) control, delay, and packet rate for the first user's bearer-0 (B0) flow in the MGW(2,16) pipeline, delay-SLO fixed at 1 msec, bearer-0 rate varying between 1 kpps and 50 kpps for all users.

test Batchy with variable bitrate flows. Intuitively, when the offered load drops abruptly it suddenly takes much longer for a buffer to accumulate enough packets to construct a batch, which causes delay to skyrocket and thereby leads to a grave delay-SLO violation. In such cases Batchy needs to react fast to recover. On the other hand, when the packet rate increases and queuing delays fall, Batchy should gradually increase triggers across the pipeline to improve batch-efficiency.

To understand the system dynamics under changing packet rates, we conducted two experiments. First, we fire up the VRF(4,8) pipeline and we fix the delay-SLO for all flows at 1 msec and vary the total offered load between 50 kpps to 3 mpps in 6 steps; this amounts to a dynamic range of 3 orders of magnitude (see `conf/l2l3_vbr.batchy` in [4]). Second, we set up the MGW(2,16) pipeline (2 bearers and 16 users, see `conf/mgw_vbr.batchy` in [4]), but now only bearer-0 flows (B0, the "QoS" bearer) vary the offered packet rate (between 1 kpps and 50 kpps) and set a delay-SLO (again, 1 msec). The results are in Fig. 9b and Fig. 9c, respectively.

Our observations here are similar as before. Even in the face of widely changing packet rates, Batchy keeps delay firmly below 1 msec except under transients: it *instantaneously recovers from SLO violations and rapidly returns to operate at full batch-size whenever possible*. Meanwhile, the max-controller causes a 100× SLO violation at small packet rates. In the second experiment we again see significant improvement in the total throughput with Batchy, compared to the null-controller (recall, only bearer-0 rate is fixed in this case).

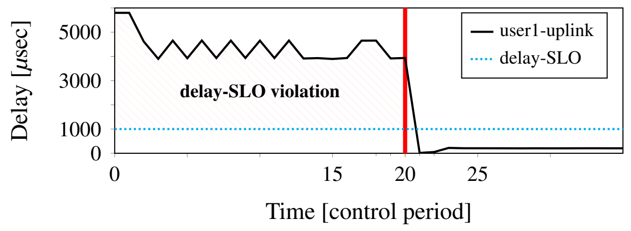


Figure 10: Recovery from inherent delay-SLO violations: MGW(2,8) test case with two users at bearer-0, delay-SLO set to 1 msec. The pipeline controller is started manually at the 20-th control period, moving bulk bearer-1 uplink and downlink traffic to a new worker each. (Downlink traffic and the other user's traffic exhibit similar performance.)

Resource-(re)allocation when SLOs cannot be satisfied. Finally, we study the effects of inherent delay-SLO violations, which occur when the turnaround time grows prohibitive at an over-provisioned worker and ingress queue latency exceeds the delay-SLO even before packets would enter the pipeline. Batchy implements a pipeline controller to detect inherent delay-SLO violations and to heuristically re-allocate resources, decomposing the data flow graph to multiple subgraphs to move delay-insensitive traffic to new workers.

Fig. 10 shows the pipeline controller in action (see `conf/mgw_decompose.batchy`) in [4]). Again we set up the

MGW(2,8) pipeline but now only two users open a flow at bearer-0, with delay-SLO set to 1 msec both in the uplink and downlink direction. The rest of the users generate bulk traffic at bearer-1 with no delay-SLO set. In addition, the service time of the bearer-1 uplink/downlink chains is artificially increased, which causes the turnaround time to surpass 1 msec and the latency for the delay-sensitive bearer-0 flows to jump to 4 msec. The pipeline controller kicks in at the 20-th control period and *quickly recovers the pipeline from the inherent delay-SLO violation*: by decomposing the data flow graph at the output of the `Bearer selector` splitter module, it moves all bearer-1 traffic away to new workers. The delay of bearer-0 flows quickly falls below the SLO, so much so that from this point Batchy can safely increase buffer sizes across the pipeline, leading to more than 10× improvement in the cumulative throughput (not shown in the figure).

6 Related Work

Batch-processing in data-intensive applications. Earlier work hints at the dramatic performance improvement batch-processing may bring in data-intensive applications and in software packet I/O in particular [1,6,7,20,29,41,50]. Consequently, batch-based packet-processing has become ubiquitous in software network switches [2,3,14,15,20,27,30,36], OS network stacks and dataplanes [3,6,8,11], user-space I/O libraries [1,16], and Network Function Virtualization [19,21,42,45,50]. Beyond the context of performance-centric network pipelines, batch-processing has also proved useful in congestion control [37], data streaming [18], analytics [44], and machine-learning [10].

Dynamic batch control. Clearly, the batch size should be set as high as possible to maximize performance [1,6,8,11,16,20,27,29,41,50]; as long as I/O rates are in the range of multiple million packets per second the delay introduced this way may not be substantial [16]. Models to compute the optimal batch size *statically* and *globally* for the entire pipeline, subject to given delay-SLOs, can be found in [6,22,41,50]; [41] presents a discrete Markovian queuing model and [22] presents a discrete-time model for a single-queue single-server system (c.f. Fig. 4) with known service-time distribution. *Dynamic* batch-size control was proposed in [29], but again this work considers packet I/O only. Perhaps the closest to Batchy is IX [6], which combines run-to-completion and batch-size optimization to obtain a highly-efficient OS network data-plane, and NBA [20], which observes the importance avoiding "the batch split problem" in the context of data flow graph scheduling. Batchy extends previous work by providing a unique combination of dynamic *internal batch de-fragmentation* (instead of applying batching only to packet I/O), analytic techniques for *controlling queue backlogs* (using a new abstraction, fractional buffers), and *selective SLO-enforcement* at the granularity of individual service chains (extending batching to bulk flows even in the presence of delay-sensitive traffic).

Data flow graph scheduling. Data flow graphs are universal in data-intensive applications, like multimedia [43], machine learning [10], and robot control [39]. However, most of the previous work on graph scheduling considers a different context: in [23,33] the task is to find an optimal starting time for the parallel execution of processing nodes given dependency chains encoded as a graph, while [13] considers the version of the scheduling problem where graph-encoded dependencies exist on the input jobs rather than on the processing nodes. Neither of these works takes batch-processing into account.

Service chains and delay-SLOs. With network function virtualization [26] and programmable software switches [24,38] becoming mainstream, scheduling in packet-processing systems has received much attention lately. Previous work considers various aspects of NF scheduling, like parallel [42] implementation on top of process schedulers [21], commodity data-centers [19,35], and run-to-completion frameworks in an isolated manner [36], or in hybrid CPU–GPU systems [15,20,48,49]. Recent work also considers SLOs: [45] uses CPU cache isolation to solve the noisy neighbor problem while [50] extends NFV to GPU-accelerated systems and observes the importance of controlling batch-size to enforce delay-SLOs. Apart from complementing these works, Batchy also contributes to recent effort on network function performance profiling (BOLT [17]), efficient worker/CPU resource allocation for enforcing delay-SLOs (Shenango [34]), and avoiding cross-CPU issues in NF-scheduling (Metron [19]).

7 Conclusions

In this paper we introduce Batchy, the first general-purpose data flow graph scheduler that makes batch-based processing a first class citizen in delay-sensitive data-intensive applications. Using a novel batch-processing profile and an analytic performance modeling framework, Batchy balances batch-processing efficiency and latency and delivers strict SLO-compliance at the millisecond scale even at multiple millions of packets per seconds of throughput. As such, Batchy could be used as a central component in 5G mobile cores (e.g., the MGW use case) and industry-automation (e.g., the robot-controller use case) applications and latency-optimized network function virtualization (e.g., the VRF use case). It may also find use outside the networking context, as the batch-scheduler in streaming, analytics, machine learning, or multimedia and signal processing applications. In these applications, however, the default run-to-completion execution model adopted in Batchy may not provide sufficient workload isolation guarantees; future work therefore involves implementing a WFQ controller based on the model (8)–(10) to incorporate Batchy into an explicit process-scheduling model.

References

[1] Advanced Networking Lab/KAIST. Packet I/O Engine. `https:`

//github.com/PacketShader/Packet-IO-Engine.

[2] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi. High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine*, 56(12):97–103, December 2018.

[3] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *ACM/IEEE ANCS*, pages 5–16, 2015.

[4] Batchy. https://github.com/hsnlab/batchy.

[5] Mokhtar S Bazaraa, Hanif D Sherali, and Chitharanjan M Shetty. *Nonlinear programming: Theory and algorithms*. John Wiley & Sons, 2013.

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX OSDI*, pages 49–65, 2014.

[7] Ankit Bhardwaj, Atul Shree, V. Bhargav Reddy, and Sorav Bansal. A Preliminary Performance Model for Optimizing Software Packet Processing Pipelines. In *ACM APSys*, pages 26:1–26:7, 2017.

[8] Jesper Dangaard Brouer. Network stack challenges at increasing speeds. Linux Conf Au, Jan 2015.

[9] The CAIDA UCSD Anonymized Internet Traces - 2019. Available at http://www.caida.org/data/passive/passive_dataset.xml, 2019.

[10] Hong-Yunn Chen et al. TensorFlow: A system for large-scale machine learning. In *USENIX OSDI*, volume 16, pages 265–283, 2016.

[11] Jonathan Corbet. Batch processing of network packets. Linux Weekly News, Aug 2018.

[12] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *ACM SIGCOMM*, pages 27–38, 2014.

[13] Mark Goldenberg, Paul Lu, and Jonathan Schaeffer. TrellisDAG: A system for structured DAG scheduling. In *JSSPP*, pages 21–43, 2003.

[14] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[15] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A GPU-accelerated software router. In *ACM SIGCOMM*, pages 195–206, 2010.

[16] Intel. Data Plane Development Kit. http://dpdk.org.

[17] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. Performance contracts for software network functions. In *USENIX NSDI*, pages 517–530, 2019.

[18] Apache Kafka. https://kafka.apache.org.

[19] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX NSDI*, pages 171–186, 2018.

[20] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (Network Balancing Act): A high-performance packet processing framework for heterogeneous processors. In *EuroSys*, pages 22:1–22:14, 2015.

[21] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *ACM SIGCOMM*, pages 71–84, 2017.

[22] S. Lange, L. Linguaglossa, S. Geissler, D. Rossi, and T. Zinner. Discrete-time modeling of NFV accelerators that exploit batched processing. In *IEEE INFOCOM*, pages 64–72, April 2019.

[23] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, June 1991.

[24] T. Lévai, G. Pongrácz, P. Megyesi, P. Vörös, S. Laki, F. Németh, and G. Rétvári. The price for programmability in the software data plane: The vendor perspective. *IEEE Journal on Selected Areas in Communications*, 36(12):2621–2630, December 2018.

[25] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *ACM SIGCOMM*, pages 1–14, 2016.

[26] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco, M. Jarschel, and G. Bianchi. Survey of performance acceleration techniques for network function virtualization. *Proceedings of the IEEE*, pages 1–19, 2019.

[27] Leonardo Linguaglossa, Dario Rossi, Salvatore Pontarelli, Dave Barach, Damjan Marjon, and Pierre Pfister. High-speed software data plane via vectorized packet processing. *Tech. Rep.*, 2017. https://perso.telecom-paristech.fr/drossi/paper/vpp-bench-techrep.pdf.

[28] Nesredin Mahmud et al. Evaluating industrial applicability of virtualization on a distributed multicore platform. In *IEEE ETFA*, 2014.

[29] M. Miao, W. Cheng, F. Ren, and J. Xie. Smart batching: A load-sensitive self-tuning packet I/O using dynamic batch sizing. In *IEEE HPCC*, pages 726–733, Dec 2016.

[30] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. Dataplane specialization for high-performance OpenFlow software switching. In *ACM SIGCOMM*, pages 539–552, 2016.

[31] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *ACM SOSP*, pages 217–231, 1999.

[32] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, RFC Editor, January 1984.

[33] T. W. O'Neil, S. Tongsima, and E. H. Sha. Extended retiming: optimal scheduling via a graph-theoretical approach. In *IEEE ICASSP*, volume 4, 1999.

[34] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX NSDI*, pages 361–378, 2019.

[35] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for NFV applications. In *ACM SOSP*, pages 121–136, 2015.

[36] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *USENIX OSDI*, pages 203–216, 2016.

[37] Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. Flowtune: Flowlet control for datacenter networks. In *USENIX NSDI*, pages 421–435, 2017.

[38] Ben Pfaff et al. The design and implementation of Open vSwitch. In *USENIX NSDI*, pages 117–130, 2015.

[39] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[40] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient NIC packet scheduling. In *USENIX NSDI*, pages 33–46, February 2019.

[41] Z. Su, T. Begin, and B. Baynat. Towards including batch services in models for DPDK-based virtual switches. In *GIIS*, pages 37–44, 2017.

[42] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. NFP: enabling network function parallelism in NFV. In *ACM SIGCOMM*, pages 43–56, 2017.

[43] Wim Taymans, Steve Baker, Andy Wingo, Ronald S. Bultje, and Stefan Kost. *GStreamer Application Development Manual*. Samurai Media Limited, United Kingdom, 2016.

[44] The Apache Spark project. Setting the Right Batch Interval. `https://spark.apache.org/docs/latest/streaming-programming-guide.html#setting-the-right-batch-interval`.

[45] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in network function virtualization. In *USENIX NSDI*, pages 283–297, 2018.

[46] Carl A Waldspurger and E Weihl W. Stride scheduling: deterministic proportional-share resource management, 1995. Massachusetts Institute of Technology.

[47] Ed Warnicke. Vector Packet Processing - One Terabit Router, July 2017.

[48] Xiaodong Yi, Jingpu Duan, and Chuan Wu. GPUNFV: A GPU-accelerated NFV system. In *ACM APNet*, pages 85–91, 2017.

[49] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. G-NET: Effective GPU sharing in NFV systems. In *USENIX NSDI*, pages 187–200, 2018.

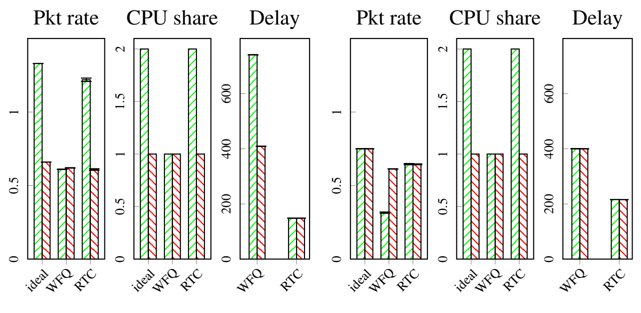
[50] Zhilong Zheng, Jun Bi, Haiping Wang, Chen Sun, Heng Yu, Hongxin Hu, Kai Gao, and Jianping Wu. Grus: Enabling latency SLOs for GPU-accelerated NFV systems. In *IEEE ICNP*, pages 154–164, 2018.

Appendix

A Data Flow Graph Scheduling

Scheduling in the context of data flow graphs means to decide which module to execute next. The goal of the scheduler is to provide efficiency and fairness: efficiency is ultimately determined by the amount of load the system can process from the ingress modules to the egress modules and fairness is generally measured by the extent to which the eventual resource allocation is *rate-proportional* [21]. Here, limited CPU resources need to be allocated between competing modules based on the combination of the offered load (or arrival rate) for the module and its processing cost. Intuitively, if either one of these metrics is fixed then the CPU allocation should be proportional to the other metric. Consider the example in Fig. 2; if the two modules have the same CPU cost but NF1 has twice the offered load than NF2, then we want it to have twice the CPU time allocated, and hence twice the output rate, relative to NF2. Alternatively, if the NFs have the same offered load but NF1 incurs twice the processing cost then we expect it to get twice as much CPU time, resulting in both modules having roughly the same output rate.

Explicit scheduling. In explicit scheduling there is a standalone mechanism that runs side-by-side with the pipeline and executes modules in the given order. A typical example is Weighted Fair Queueing (WFQ) or Completely Fair Scheduling (CFS), where the user assigns integer *weights* to



(a) Asymmetric rate (b) Asymmetric cost

Figure 11: Rate-proportional fairness in WFQ and run-to-completion scheduling in the *asymmetric rate* case (NF1 receives twice the offered load of NF2 and CPU costs are equal) and *asymmetric cost* case (same offered load but NF1 needs twice as much CPU time to process a packet as NF2). Packet rate is in mpps and delay is in μsec, and 🟩 denotes the first flow while 🟥 denotes the second flow as in Fig. 2.

Pipeline	# modules	Batch size [pkt]					Rate [Mpps]					Delay [% of violations]				
		Null	Max	NFVnice	Batchy/on-off	Batchy/full	Null	Max	NFVnice	Batchy/on-off	Batchy/full	Null	Max	NFVnice	Batchy/on-off	Batchy/full
L2L3(1)	9	31.5	32	31.8	31.5	31.5	8.29	8.31	4.78	8.29	8.47	0%	100%	100%	0%	0%
L2L3(4)	21	12.6	32	14.53	12.6	29.2	7.49	7.46	5.86	7.26	7.77	0%	100%	13%	0%	0%
L2L3(8)	37	7.0	32	8.1	5.8	30.9	6.26	7.38	4.65	5.18	7.42	0%	100%	0%	0%	0%
L2L3(16)	69	2.6	32	3.92	3.7	28.9	3.63	6.11	2.75	4.94	6.33	0%	100%	0%	0%	0.5%
*L2L3(16)	69	8.5	32	4.7	9.2	24.2	5.24	5.7	2.4	5.25	5.79	0%	75%	2%	8.3%	2.4%
*L2L3(32)	133	3.1	32	3	5.8	24.5	4.47	5.32	1.46	4.61	5.48	0%	63%	0%	1.9%	1.3%
*L2L3(64)	261	1.1	32	2.1	4.9	23.4	2.12	5.17	0.82	3.81	5.03	0%	68%	1%	5.7%	5.2%
GW(1)	12	31.5	32	31.7	31.5	31.5	6.06	6.08	3.11	6.08	6.08	0%	100%	100%	0%	0%
GW(8)	61	5.6	32	6.21	5.6	31.4	4.2	5.22	2.7	4.23	5.25	0%	100%	23.6%	0%	0.6%
GW(16)	117	2.9	32	3.16	2.9	30.7	2.66	4.51	1.47	2.8	4.58	0%	100%	44.3%	0%	0.5%
GW(64)	453	2.2	32	2	2.2	24.4	1.85	3.39	0.54	1.86	3.73	0%	100%	0%	0%	5.2%
VRF(1,1)	11	32	32	32	32	32	5.88	5.76	3.48	5.87	5.92	0%	100%	100%	0%	0%
VRF(2,4)	43	5.3	32	7.1	10.7	22.2	3.49	4.77	2.53	4.37	4.03	0%	100%	54%	0%	2.2%
VRF(16,4)	323	2	32	2	10.8	24.4	1.01	2.7	0.42	2.41	2.84	0%	100%	0%	0%	8.9%
VRF(12,8)	435	1.7	32	1.7	5.9	23.4	0.95	2.34	0.35	1.82	2.5	0%	100%	0%	0%	8.2%
MGW(2,4)	110	4.5	32	4.5	13.7	17.3	1.75	3.15	1.03	1.78	2.03	0%	100%	100%	0%	0%
MGW(4,4)	206	4	32	3.7	11.4	20.2	1.5	2.83	0.8	1.69	2.19	0%	100%	61.6%	0%	1%
MGW(8,4)	398	5.1	32	4.1	20.9	25.7	1.61	2.77	0.6	2.12	2.5	0%	100%	0%	0%	5%
MGW(16,4)	782	4.9	32	4.1	25.9	27.8	1.49	2.34	0.39	2.1	2.25	0%	100%	0%	0%	0.3%
RC(16)	25	2.24	32	3.43	30.14	30.14	0.43	3.27	0.58	3.25	3.27	100%	100%	100%	0%	0%

Table 2: Static evaluation results with the null-controller, max-controller, NFVnice, Batchy/on-off and Batchy/full on different pipelines: number of modules, average batch-size over the pipeline, packet rate, and delay statistics in terms of the percentage of control periods when a delay-SLO violation was detected for at least one flow.

modules and the scheduler ensures that the runtime resource allocation will be proportional to modules' weight. WFQ does not provide rate-proportional fairness out of the box; e.g., in the example of Fig. 2 NF1 will *not* receive more CPU time neither when its offered load (asymmetric rate) or processing cost (asymmetric cost) is twice that of NF2 (see Fig. 11). Correspondingly, WFQ schedulers need substantial tweaking to approximate rate-proportional fairness, and need further optimization to avoid head-of-line blocking and late drops along a service chain [21]. Even running the scheduler itself may incur non-trivial runtime overhead. Worse still, packets may get dropped *inside* the pipeline when internal queues overflow; this may be a feature (e.g., when we want to apply rate-limitation or traffic policing via the scheduler) or a bug (when useful traffic gets lost at an under-provisioned queue).

Run-to-completion execution. This model eliminates the explicit scheduler and its runtime overhead all together. In run-to-completion execution the entire input batch is traced though the data flow graph in one shot, by upstream modules automatically scheduling downstream modules whenever there is work to be done [6, 14]. As Fig. 11 shows, this model introduces much smaller delay vs. explicit scheduling, as it needs no internal queues. In addition, run-to-completion provides rate-proportional fairness out-of-the-box, even without additional tweaking and without the risk of head-of-line blocking and internal packet drops. This yields an appealingly simple "schedulerless" design. On the other hand, since module execution order is automatically fixed by the pipeline and the scheduler cannot by itself drop packets, the share of CPU time a module gets is determined by the offered load only. This makes enforcing rate-type SLOs through a run-to-completion scheduler difficult. In our example, if NF2 receives twice the packet rate of NF1 then it will receive twice the CPU share, and hence the second flow will have twice the output rate, even though we may want this to be the other way around.

B Static Evaluations: Detailed Results

The detailed static performance results are given in Table 2. The table specifies the number of modules in each pipeline, the batch size statistics averaged across each module in time, the throughput as the cumulative packet rate summed over all flows, and the delay-SLO violation statistics as the percentage of control periods when we detected an SLO violation for at least one flow, for each of the 5G NF use cases, varying pipeline complexity using different settings for the parameters n and m.

C The Fractional Buffer

The below algorithm summarizes the execution model of a fractional buffer. Here, *queue* means the internal queue of the fractional buffer, b is the trigger, *q.pop(x)* dequeues x packets from the beginning of queue q, and $q.push(batch)$ appends the packets of $batch$ to the queue q.

```
procedure FRACTIONALBUFFER::PUSH($batch$)
    while $queue.size \leq b$ AND $batch.size > 0$ do
        $queue.push(batch.pop(1))$
    end while
    if $queue.size = b$ then
        $new\_batch \leftarrow queue.pop(b)$
        process $new\_batch$ through the network function
        put $v$'s downstream modules to the run queue
        $queue.push(batch.pop(batch.size))$
    end if
end procedure
```

D The Projected Gradient Controller

Below, we discuss the high-level ideas in the projected gradient controller implemented in Batchy and then we give the detailed pseudocode.

First, we compute the objective function gradient $\nabla l =$

$[\partial l/\partial b_v : v \in V]$, which measures the sensitivity of the total system load $l = \sum_{v \in V} l_v$ as of (11) to small changes in the trigger b_v for each module:

$$\frac{\partial l}{\partial b_v} = -\frac{\tilde{r}_v T_{0,v}}{b_v^2} = -\frac{\tilde{x}_v T_{0,v}}{b_v} \quad .$$

The delay-gradients $\nabla t_f = [\partial t_f/\partial b_v : f \in \mathcal{F}]$ are as follows:

$$\frac{\partial t_f}{\partial b_v} = \begin{cases} \frac{1}{\tilde{r}_v} + T_{1,v} & \text{if } v \in p_f \\ 0 & \text{otherwise} \end{cases}$$

Note that the delay t_f of a flow f is affected only by the modules along its path p_f, as long as the turnaround time is considered constant as of (3).

Second, project the objective gradient ∇l to the feasible (i.e., SLO-compliant) space. For this, identify the flows f that may be in violation of the delay-SLO: $\tilde{t}_f \geq (1-\delta)D_f$.

Third, let M be a matrix with row i set to ∇t_f if f is the i-th flow in delay violation and compute the projected gradient $\Delta b = -(I - M^T(MM^T)^{-1}M)\nabla l$. Note that if M is singular or the projected gradient becomes zero then small adjustments need to be made to the projection matrix [5].

Fourth, perform a line-search along the projected gradient Δb. If for some module v the corresponding projected gradient component Δb_v is strictly positive (it cannot be negative) then calculate the largest possible change in b_v that still satisfies the delay-SLO of *all* flows traversing v:

$$\lambda_v = \min_{f \in \mathcal{F}: v \in p_f} \frac{D_f - \tilde{t}_f}{\Delta b_v} \quad .$$

Finally, take $\lambda = \min_{v \in V} \lambda_v$ and adjust the trigger of each module v to $b_v + \lceil \Delta b_v \lambda \rceil$. Rounding the trigger up to the nearest integer yields a more aggressive control.

The below pseudo-code describes the projected gradient controller in detail. Vectors and matrices are typeset in bold in order to simplify the distinction from scalars. We generally substitute matrix inverses with the Moore-Penrose inverse in order to take care of the cases when **M** is singular.

```
procedure ProjectedGradient($\mathcal{G}, \mathcal{F}, D, f$)
    M is a matrix with row $i$ set to $\nabla t_f = \left[\frac{\partial t_f}{\partial b_v} : f \in \mathcal{F}\right]$, where $f$
is the $i$-th flow in $\mathcal{F}$ with $\tilde{t}_f \geq (1-\delta)D_f$
    ▷ Gradient projection
    while True do
        $\mathbf{P} = \mathbf{I} - \mathbf{M}^T(\mathbf{M}\mathbf{M}^T)^{-1}\mathbf{M}$
        $\Delta\mathbf{b} = \mathbf{P}\nabla\mathbf{l}$
        if $\Delta\mathbf{b} \neq 0$ then break
        $\mathbf{w} = -(\mathbf{M}\mathbf{M}^T)^{-1}\mathbf{M}\nabla\mathbf{l}$
        if $\mathbf{w} \geq 0$ then return          ▷ Optimal KKT point reached
        delete row for $f$ from M for some $f \in \mathcal{F} : w_f < 0$
    end while
    ▷ Line search
    for $v \in V, f \in p_v$ do
        if $\Delta b_v > 0$ then
            $\lambda_v = \min_{f \in \mathcal{F}: v \in p_f} \left\lceil \frac{D_f - \tilde{t}_f}{\Delta b_v} \right\rceil$
        end if
    end for
    $\lambda = \min_{v \in V} \lambda_v$
    for $v \in V$ do SetTrigger($v$, $b_v + \Delta b_v \lambda$)
end procedure
```

E The Feasibility-recovery Algorithm

The projected gradient controller cannot by itself recover from situations when a fractional buffer at some module is triggered at a too small rate to deliver the required delay-SLO to each flow traversing the module. The below pseudo-code describes the feasibility recovery process implemented in Batchy, which is implemented to handle such situations.

```
procedure FeasibilityRecovery($\mathcal{G}, \mathcal{F}, D, f$)
    for $f \in \mathcal{F}$ do $t_f \leftarrow \tilde{t}_f$
    ▷ Recover from SLO violation
    for $v \in V : b_v \geq \tilde{b}_v^{in}$ do
        if $\exists f \in \mathcal{F} : v \in p_f$ AND $t_f \geq (1-\varepsilon)D_f$ then
            $\Delta b_v = \max_{f \in \mathcal{F}: v \in p_f \wedge t_f \geq (1-\varepsilon)D_f} \left\lceil \frac{D_f - t_f}{\partial t_f/\partial b_v} \right\rceil$
            if $\Delta b_v > b_v - \tilde{b}_v^{in}$ then $\Delta b_v \leftarrow b_v - \tilde{b}_v^{in}$
            for $f \in \mathcal{F} : v \in p_f$ do $t_f \leftarrow t_f - \frac{\partial t_f}{\partial b_v}\Delta b_v$
            $b_v \leftarrow b_v - \Delta b_v$
        end if
    end for
    ▷ Injecting a buffer
    for $v \in V : b_v = 0$ do
        if $\forall f \in \mathcal{F} : v \in p_f$ it holds that $t_f < (1-\varepsilon)D_f$ then
            $\Delta b_v = \min_{f \in \mathcal{F}: v \in p_f \wedge t_f < (1-\varepsilon)D_f} \left\lceil \frac{t_f - D_f}{\partial t_f/\partial b_v} \right\rceil$
            if $\Delta b_v > \tilde{b}_v^{in}$ then $\Delta b_v \leftarrow \tilde{b}_v^{in}$
            for $f \in \mathcal{F} : v \in p_f$ do $t_f \leftarrow t_f + \frac{\partial t_f}{\partial b_v}\Delta b_v$
            $b_v \leftarrow \Delta b_v$
        end if
    end for
    for $v \in V$ do SetTrigger($v$, $b_v$)
end procedure
```

F The Data Flow Graph Decomposition Algorithm

Pipeline decomposition is initiated in Batchy whenever an inherent delay-SLO violation is detected. This occurs when a worker is overprovisioned and the turnaround time grows beyond the SLO for a delay-sensitive flow; in such cases Batchy migrates delay-insensitive traffic to new workers to address SLO violations. The below pseudo-code describes the pipeline decomposition procedure implemented in Batchy.

```
procedure DecomposePipeline($\mathcal{G}, \mathcal{F}, D, f$)
    $V_t \leftarrow \emptyset$
    $F_t \leftarrow \emptyset$
    $\tau_t \leftarrow \emptyset$
    for $f \in (F$ in ascending order of $D_f)$ do
```

```
        if CHECK_DELAY_SLO($V_t, F_t, \tau_t, f$) then
            $F_t \leftarrow F_t \cup f$
            $V_t \leftarrow V_t \cup p_f$
            $\tau_t \leftarrow \tau_t + \sum_{v \in P_f, v \notin V_t} (T_{0v} + T_{1v} B)$
        else
            $M \leftarrow \emptyset$
            MIGRATEFLOWS($V_t, F_t, M, \mathcal{F}$)
        end if
    end for
end procedure
procedure MIGRATEFLOWS($V_t, F_t, M, \mathcal{F}$)
    for $g \in (\mathcal{F} \setminus F_t)$ do
        for $v \in p_g$ do
            if $v \notin V_t$ and $v \notin M$ then
                create new worker
                add a queue before $v$
                attach queue to new worker
                $M \leftarrow M \cup v$
            end if
        end for
    end for
end procedure
procedure CHECKDELAYSLO($V_t, F_t, \tau_t, f$)
    for $g \in (F_t \cup f)$ do
        if $(\tau_t + \sum_{v \in P_f, v \notin V_t} (T_{0v} + T_{1v} B) + \sum_{v \in P_g} (T_{0v} + T_{1v})) > D_g$
then
            return False
        end if
        return True
    end for
end procedure
```

Adapting TCP for Reconfigurable Datacenter Networks

Matthew K. Mukerjee[⋆†], Christopher Canel[⋆], Weiyang Wang[∘], Daehyeok Kim[⋆‡], Srinivasan Seshan[⋆], Alex C. Snoeren[∘]
[⋆]*Carnegie Mellon University,* [∘]*UC San Diego,* [†]*Nefeli Networks,* [‡]*Microsoft Research*

Abstract

Reconfigurable datacenter networks (RDCNs) augment traditional packet switches with high-bandwidth reconfigurable circuits. In these networks, high-bandwidth circuits are assigned to particular source-destination rack pairs based on a schedule. To make efficient use of RDCNs, active TCP flows between such pairs must quickly ramp up their sending rates when high-bandwidth circuits are made available. Past studies have shown that TCP performs well on RDCNs with millisecond-scale reconfiguration delays, during which time the circuit network is offline. However, modern RDCNs can reconfigure in as little as 20 μs, and maintain a particular configuration for fewer than 10 RTTs. We show that existing TCP variants cannot ramp up quickly enough to work well on these modern RDCNs. We identify two methods to address this issue: First, an in-network solution that dynamically resizes top-of-rack switch virtual output queues to prebuffer packets; Second, an endpoint-based solution that increases the congestion window, `cwnd`, based on explicit circuit state feedback sent via the `ECN-echo` bit. To evaluate these techniques, we build an open-source RDCN emulator, Etalon, and show that a combination of dynamic queue resizing and explicit circuit state feedback increases circuit utilization by 1.91× with an only 1.20× increase in tail latency.

1 Introduction

Modern datacenter applications need high-bandwidth, high–port-count, low-latency, low-cost networks to connect their hosts. Unfortunately, traditional packet switches are hitting CMOS manufacturing limits and are unable to simultaneously provide both high bandwidth and large numbers of ports [43]. Thus, researchers have proposed augmenting datacenter networks with reconfigurable circuit switches (e.g., optical or wireless) that provide high bandwidth between racks *on demand* [6, 16, 20, 25, 26, 32, 38, 42, 47, 51, 57].

However, it can be challenging for endpoints to extract the full potential of *reconfigurable datacenter networks* (*RDCNs*) that combine both circuit and packet networks. Circuit switches incur non-trivial reconfiguration delays while they adjust the high-bandwidth topology, and portions of the circuit network may be unavailable during these periods. Hence, such hybrid designs often result in fluctuations between periods of high bandwidth—when a circuit is provisioned—and low bandwidth—when the packet network is in use. While periods of higher bandwidth are attractive in principle, recent proposals suggest adjusting the topology frequently. The resulting bandwidth fluctuations pose a problem for end-host applications: their active TCP connections must rapidly increase transmission rates to use the available bandwidth and then slow down again to avoid massive queuing. In this paper, we explore these adverse interactions between TCP and RDCNs, and techniques to mitigate their performance impacts.

To amortize the cost of reconfiguration, circuits must be provisioned for a long period of time relative to the reconfiguration delay of the switch. TCP interactions with RDCNs were initially explored in the context of switches with *millisecond*-scale reconfiguration delays [16, 51]. Given the sub-millisecond propagation delays found in modern datacenters, circuits in millisecond-scale reconfiguration networks are enabled for many, many round-trip times (RTTs). These early studies found that TCP was able to adapt to the link speed changes over these time periods. However, modern reconfigurable switches that can change link state on the scale of *microseconds* [20, 38, 47] have redefined these problems. At first glance, these lower reconfiguration delays result in lower overheads and allow more rapid provisioning of bandwidth to where it is needed. However, when circuit uptimes are only a few (e.g., <10) RTTs long, TCP's ramp-up is too slow to utilize the large (e.g., 10×), temporary bandwidth increase before the circuit disappears, leading to low circuit utilization (Section 3). This raises the question of whether an RDCN can employ the rapid reconfigurations needed to meet changing traffic demands while also providing good performance.

The performance issues arise from a broken assumption made by end-hosts about the network: congestion control algorithms generally assume that bandwidth does not fluctuate at short timescales. We explore the consequences of this broken assumption for TCP, and identify two methods, at different levels in the network stack, for ramping up TCP in the environment of rapid bandwidth fluctuation.

First, we build on the insight that in RDCNs, bandwidth fluctuation is not arbitrary; it is part of a known schedule. Therefore, we can proactively modify the network to transparently influence end-host behavior. In this case, we entice TCP to ramp up earlier by eliminating packet drops due to full top-of-rack (ToR) switch virtual output queues (VOQs), thereby triggering end-hosts to increase their sending rates. We accomplish this by dynamically increasing the size of ToR VOQs in advance of provisioning a circuit (Section 5.2). Dynamic VOQ resizing does not require modifying end-hosts and, thus, works with existing TCP implementations.

Our second technique involves minor modifications to the end-host TCP stack to enable further performance improvements. At sending end-hosts, we increase the congestion window, `cwnd`, based on explicit circuit state feedback sent by the reconfigurable switch (Section 5.3). For some rack pair (S, D), we configure our emulated switch to set the `ECN-echo` (ECE) bit in the TCP headers of ACKs sent by D if there is currently a circuit enabled from S to D . The sender monitors the ECE stream, explicitly expanding and contracting `cwnd` when circuits begin and end, respectively.

To evaluate our solutions, we design and implement an open-source RDCN emulator, *Etalon*[1], for use on public testbeds (Section 4). Experiments on 3-rack (48-host) and 8-rack (128-host) emulated testbeds show that dynamic buffer resizing and explicit circuit state feedback increase circuit utilization by 1.91× while increasing 99th percentile tail latency by 1.20× (Sections 6.1 and 6.3).

Ultimately, datacenters must adapt to reap the benefits of RDCNs. In-network changes yield impressive improvements, and if modifying end-hosts is an option, then even higher performance is feasible. We make three contributions:

1. We characterize the critical challenge of rapid bandwidth fluctuation in RDCNs. We use a combination of experimental results and simulations to identify the range of reconfiguration delays that impact TCP performance, and show that a wide range of TCP congestion control algorithms suffer from poor performance in these settings.
2. We propose two solutions, at different layers of the network stack, to ramp up TCP under rapid bandwidth fluctuation: dynamic buffer resizing and explicit circuit state feedback. Our evaluation of these techniques shows the benefits that modifying higher network layers can have for RDCNs.
3. We design and implement an emulation platform, Etalon, for evaluating hybrid networks end-to-end with real applications, and use it to demonstrate the efficacy of our proposed techniques. Etalon is open source [15].

[1]Named after an optical filter used for solar observation.

2 Background

To better understand the challenges and solutions presented in this paper, we first examine RDCNs in detail in Figure 1. While we use optical circuit switching [16, 38, 47, 51] as an illustrative example for the rest of the paper, the results generalize to other reconfigurable technologies, such as free-space optics [20, 26] and 60-GHz wireless [25, 32, 57]). We eschew older millisecond-scale reconfigurable switches [16, 51] for modern microsecond-scale switches [20, 38, 42, 47], as the nature of the challenges and solutions differ with timescale.

2.1 Hybrid Network Model

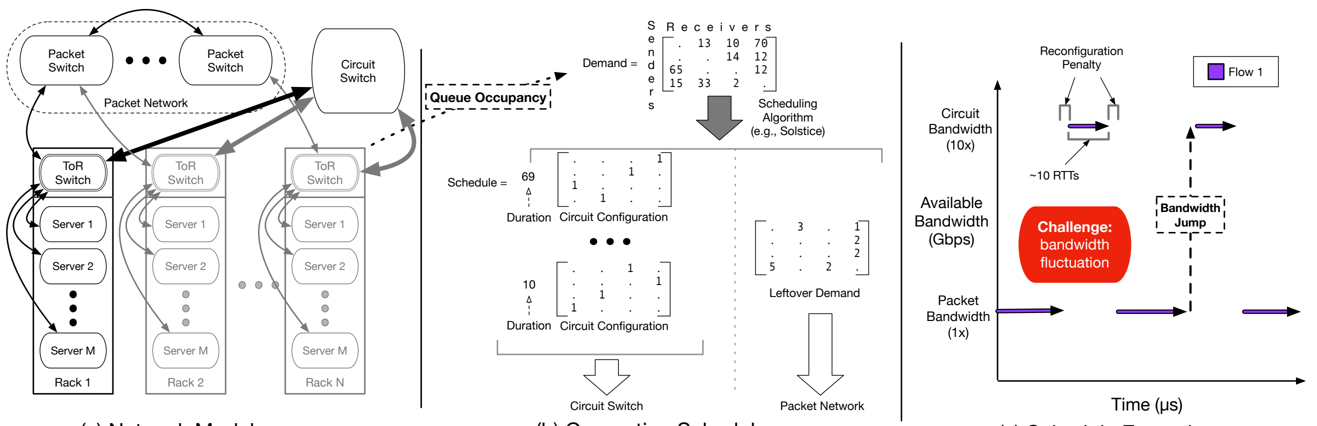
We consider an RDCN of N racks of M servers, with each rack containing a ToR switch (Figure 1(a)). ToRs connect racks to an arbitrarily-complex packet network (one or more switches) and a circuit network composed of a single, central circuit switch. The packet network is low bandwidth (e.g., 10 Gb/s), but can make forwarding decisions for individual packets. The circuit network is high bandwidth (e.g., 80-100 Gb/s), but makes forwarding decisions for many packets on much longer timescales to amortize its reconfiguration penalty.

Reconfiguration time is an inherent trade-off in circuit-switched networks. Instead of providing continuous connectivity between all endpoints, like in a packet network, a circuit network establishes exclusive, but temporary, connections between *pairs* of endpoints. Here, the endpoints are the ToRs. To expand this design to provide full connectivity, the network periodically changes which pairs of endpoints are spanned by a circuit. The physical limits of the specific underlying technology determine how long this reconfiguration takes.

Following prior work [38, 39, 47], we make the pessimistic assumption that, during circuit reconfiguration, no circuit links can be used. This allows us to apply our results to a broader set of technologies. The packet network, on the other hand, can be used at all times. Both switches source packets from $N \times N$ virtual output queues (VOQs) on the ToRs. The circuit switch itself is queue-less: It functions as a crossbar, only allowing configurations that form perfect matchings [2, 16, 38, 39, 47, 51]. I.e., a given sender is connected to exactly one receiver, and vice-versa. Thus, at any point in time, the circuit switch can drain at most one VOQ on each ToR, whereas the packet switch may drain multiple VOQs on each ToR simultaneously.

2.2 Computing Circuit Schedules

The circuit scheduler is tasked with estimating demand on the network and then computing a set of configurations that best satisfies this demand (Figure 1(b)). During reconfiguration, the circuit network is unavailable. Therefore, the circuit scheduler must balance the competing objectives of (1) servicing demand from many different rack pairs by reconfiguring



(a) Network Model (b) Computing Schedules (c) Schedule Execution

Figure 1: Overview of RDCNs.

frequently and (2) achieving high circuit uptime by allowing a configuration to persist for a (relatively) long time. To achieve a high relative uptime of 90%, schedules typically hold a particular circuit state for 9× the duration of a reconfiguration.

Network scheduling in most RDCNs entails mapping rack-level demand to a set of circuit configurations (port-matchings) with corresponding time duration[2]. Any "leftover" demand is handled by the lower-bandwidth packet switch. Borrowing terminology from prior work [47], we refer to a single circuit uptime as a *day* and the reconfiguration period, during which the circuit switch is offline, as a *night*. Night length is determined by switch technology and is generally 10-30 μs [20,38,39,47]. To allow for at least 90% link uptime, the average day length must be ≥9× greater than the night length, or ~90-270 μs. A series of one or more day/night pairs that implement a set schedule is a *week*. Weeks should be sufficiently long (e.g., 2 ms) to amortize schedule computation.

Scheduling is a three-step loop: 1) Demand for the next week is estimated (e.g., through ToR VOQ occupancy); 2) An algorithm computes the schedule for the next week; 3) The schedule is disseminated to the switch. Scheduling algorithms for RDCNs (e.g., Solstice [39] and Eclipse [2]) use skew and sparsity in demand to minimize the number of configurations. Prior work on circuit scheduling informs, but is orthogonal to, our investigation into the resulting bandwidth fluctuations.

2.3 Schedule Execution

Once a schedule is disseminated to the circuit switch, it runs the circuit configurations for their respective duration (Figure 1(c)). After reconfiguration, a flow may transition from using the packet network to using the circuit network and vice versa. Because of the short day length, flows likely spend only a few (e.g., <10) RTTs on the circuit network each day. Therefore, transport protocols must cope with large (e.g., 10×) bandwidth variations based on which network a flow traverses.

[2]A major exception being RotorNet [42], which uses a predetermined schedule. It still suffers from the same bandwidth fluctuations.

Prior work has avoided the bandwidth fluctuation problem by segregating traffic into mice and elephant flows and routing them exclusively over the packet and circuit networks, respectively [16,38]. Each flow encounters one bandwidth regime, albeit the elephant flows must pause during circuit downtime. However, recent work has proven that such segregation is sub-optimal [19]. We adopt a non-segregated approach, treating the hybrid network as indivisible and routing all traffic over available circuit links, thus trading off reduced network complexity for bandwidth fluctuation.

3 TCP in RDCNs: Trends and Challenges

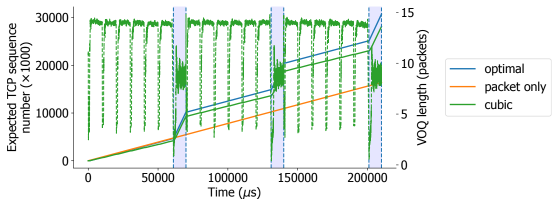
This section investigates how TCP's interactions with RDCNs are evolving with the underlying hardware trends.

3.1 Evolving Reconfiguration Delays

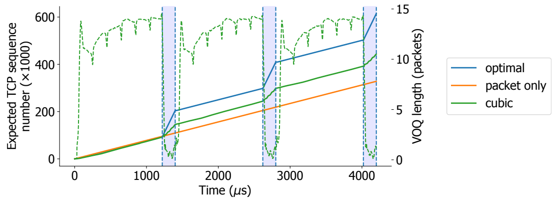
Circuit networks are characterized by their inherently high bandwidth. However, that comes at the cost of flexibility. With the exception of pathological examples, switching flexibility enables the circuit network to better serve diverse workloads. Improving flexibility by reducing the reconfiguration time has been an ongoing challenge for hardware designers, with the hope that circuit technologies will eventually be capable of approximating packet switching.

A decade ago, the best MEMS (Micro-Electro-Mechanical Systems) optical switches, which reconfigure by physically rotating laser-directing mirrors, changed paths on the order of a few milliseconds. If we generously assume that such a switch can reconfigure in 1 ms, our 90% uptime target implies that each configuration will persist for 9 ms. On network timescales, this is an eternity: Assuming a conservative RTT of 60 μs, 9 ms is 150 RTTs.

To understand how TCP behaves on RDCNs, we use an emulator, described in Section 4, and analyze the expected TCP sequence number of flows over time. We run an 8-rack schedule with 1 ms nights and 9 ms days, using the CUBIC [24]



(a) A decade ago: 1-ms reconfiguration delays and 9-ms circuit uptimes give TCP time to saturate the link.



(b) Today: TCP struggles to fill the link for circuits with short 20 μs reconfiguration delays and 180 μs uptimes.

Figure 2: TCP CUBIC performance, then and now. Circuit days are shaded in blue. Dotted lines are the corresponding VOQ length.

variant of TCP. 16 emulated hosts on rack 1 each send a flow to a counterpart host on rack 2. In this experiment, we consider a circuit switch that delivers 8× higher bandwidth than the packet network. The ToR VOQ capacity is 16 packets. The expected sequence number is measured as packets leave the hybrid switch, as described in Section 4.2, and then averaged across experimental runs.

Figure 2a shows the results of this experiment. Circuit uptimes are delineated by the blue vertical shaded regions. Since the sequence number represents the number of bytes transferred, the slope of a line corresponds directly to a flow's achieved bandwidth. We always compare to two baselines: (1) *optimal*, which is calculated based on the line rate of the packet and circuit links (taking into account that the network is offline during reconfigurations), and (2) *packet only*, which is calculated assuming that flows always use the packet network only (and that the packet network is always available). Intuitively, circuit utilization is how well the slope of a line matches that of *optimal* during the blue shaded regions. Experimental results can never exceed *optimal*, and any improvement over *packet only* illustrates the benefit of the hybrid network over a purely packet network. Throughout our analysis, we use the length of the ToR VOQs, described further in Section 4.2, to understand TCP's behavior. For a particular line on a sequence number graph, the dotted line in the same color reports the corresponding average VOQ length.

We can see that TCP roughly keeps up with the optimal throughput of the hybrid network, saturating the link. During the circuit days, which are shaded in blue, the VOQs contain approximately 8 packets. Since the VOQs never empty except for a brief moment when the days begin, there are always

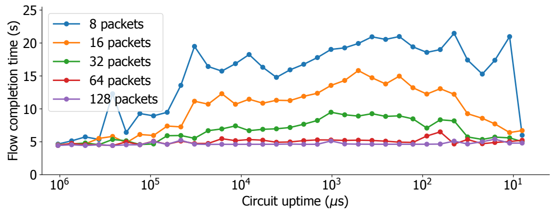


Figure 3: Simulated flow completion time for transferring 25 GB of data using 10 flows, for various buffer sizes.

packets to send over the network, resulting in high utilization.

Let us consider how hybrid networks have evolved over the past decade. Figure 2b shows the same results for a modern schedule with 20 μs nights and 180 μs days. The ratio of circuit uptime is the same, 90%, but TCP performs differently. Because the number of RTTs in a day has decreased from ~150 to ~3, TCP does not ramp up before the circuit ends. For TCP CUBIC, 3 RTTs is simply not enough time to increase the congestion window (`cwnd`) [24]. Moreover, packet drops during the subsequent reconfiguration period and the transition to the packet network cut TCP's sending rate. While TCP does recover while using the packet network, the process repeats at the next cycle, with TCP unable to ramp up to use the circuit network's full bandwidth regardless of how many periods elapse. We can also see this manifested in the VOQ length: When a day begins, the VOQs drain immediately to fill the larger bandwidth-delay product (BDP) of the circuit network, but there are insufficient outstanding packets to do so completely and insufficient time for TCP to ramp up, so the VOQs stay empty throughout the day.

To extend these results to a wider range of circuit uptimes, we ran a simulation that transfers 25 GB of data from one rack to another using 10 TCP CUBIC flows[3]. Figure 3 shows the resulting flow completion times (FCTs) for five orders of magnitude of circuit uptimes, where each line corresponds to a different amount of queuing at the ToR switch. In all cases, the RTT is 60 μs. The takeaways here are twofold: First, considering the smaller queue sizes (8-32 packets), the FCTs are low for short (e.g., 10 μs) and long (e.g., 10 ms) days, yet degrade by $2-4\times$ for moderate values (e.g., 1 ms). For short circuits, the network is effectively approximating packet switching, whereas for long circuits, the uptime is sufficient that fluctuating bandwidth is not an issue. Unfortunately, today's RDCNs fall in the middle region. Techniques to adapt TCP for RDCNs, like those presented in this paper, are necessary as long as circuit reconfiguration and propagation delays place us in a regime similar to this. We predict that the order-of-magnitude improvement in underlying circuit technology that is required to reach the "short circuits" region, where the network approximates packet switching and TCP ramp-up is no longer a problem, is still many years away.

[3]The simulator uses 1500B packets while the Etalon emulator uses 9000B packets. For the experiment in Figure 3, we scale the buffer size by 6× for easier comparison. I.e., for "8 packets", the buffer was $8 \times 6 = 48$ packets.

We find inspiration, however, from a second takeaway: With larger queue sizes (e.g., 64 and 128 packets), the flows complete quickly regardless of the circuit uptime. This indicates that large buffers build up enough excess in-flight data to burst packets quickly when more bandwidth becomes available. We discuss this further in Section 5.1, and this realization become the basis for the dynamic buffer resizing technique that we propose in Section 5.2.

3.2 Categorizing TCP Variants

Before diving into our technical solutions, it is important to remember that TCP comes in many shapes and sizes. The experiments above use TCP CUBIC, a common loss-based variant, but there are dozens of other variants designed for a plethora of network contexts, from low latency to high bandwidth to frequent loss, and more. This section gives an overview of the broad classes of TCP variants and demonstrates that no existing variant works well for hybrid networks.

At the highest level, the goal of TCP congestion control is to maximize the sending rate while fairly sharing the available bandwidth between flows and avoiding overloading the network. Determining the sending rate involves looking at signals from the network to infer its current state. The efficacy of a congestion control algorithm depends on how well it gleans information from such signals. Below, we discuss the three main categories of signals that TCP variants use to detect congestion (packet loss, network delay, and explicit network feedback), and discuss how they are effected by RDCNs.

3.2.1 Loss-based Congestion Control

Packet loss is the most commonly used congestion signal. The intuition here is that when the sender determines that packets are being lost, it assumes that the losses are a result of network congestion. Examples include TCP CUBIC [24], Reno [31], BIC [54], Illinois [40], and Highspeed [18]. In the absence of loss, these protocols increase their transmission rates to probe for available bandwidth, until a loss occurs. Different protocols choose different approaches for this probing, but all of them limit the aggressiveness of their probing to coexist reasonably with other TCP variants. This results in poor performance in RDCNs since these protocols cannot ramp up quickly enough to make use of the high-bandwidth circuits.

3.2.2 Delay-based Congestion Control

Another technique is to use measurements of the packet RTT in the congestion control protocol. Such protocols use RTT increases as an indicator of queue buildup in the network, and therefore congestion. Examples include BBR [5], TCP Vegas [4], and TIMELY [45]. Vegas and BBR are both rate-based protocols that use the difference between the offered load and the achieved throughput to detect queue buildup. TIMELY uses high-fidelity NIC timers to measure the RTT and then paces transmission based on delay gradients.

Like other TCP variants, delay-based variants are typically conservative in their probing for available bandwidth to ensure fair coexistence. An additional interaction with RDCNs is that, due to topology differences, the circuit and packet networks typically have different propagation delays (as discussed in Section 4.4: 30 μs vs. 10 μs, respectively). This poses a challenge for TCP variants that use changes in RTT as an indication of queuing.

3.2.3 Explicit Feedback–based Congestion Control

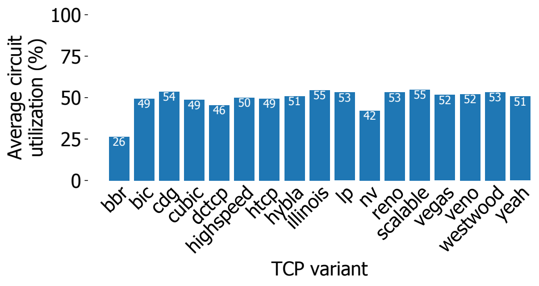
Finally, some TCP variants rely on explicit feedback from the network to detect congestion. Two manifestations of this are XCP [33] and DCTCP [1]. DCTCP responds to switch signals sent when a switch is likely to drop packets soon. If the act of a switch accepting a packet would increase its internal buffer length beyond a threshold (which is set to be lower than the total capacity of the queue), then the switch accepts the packet but sets the ECN flag in its TCP header. This flag is then communicated back to the sender via the packet's ACK. The sender monitors this stream of ECN marks to estimate when the network is close to being congested.

For domains where a single entity controls the senders and the network, coordination in this manner is a direct technique for improving performance. In Section 5.3, we propose a similar technique for adapting to bandwidth fluctuations in RDCNs that uses the ECE bit in ACKs to notify a sender when one of its flows transitions between the packet network and a high-bandwidth circuit.

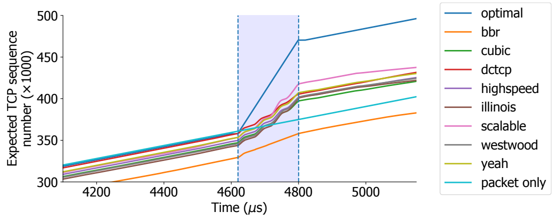
3.2.4 A Common Underlying Issue

We repeat the experiment in Section 3.1 with the 17 TCP variants pre-installed on Ubuntu 18.04. Figure 4a shows the average circuit utilization, which does not exceed 55%. Figure 4b visualizes the expected TCP sequence number over time for a selection of 8 of the 17 variants. Most perform slightly better than CUBIC, with BBR and NV falling behind, but no variant is aggressive enough to overcome the limitation that 3 RTTs is insufficient time to ramp up.

These results demonstrate that the challenge of TCP not ramping up quickly enough is not isolated to CUBIC, and cannot be solved by simply choosing a more appropriate TCP variant. Instead, we need a technique that more-directly interacts with the basic properties of TCP. Furthermore, the fact that all of the (sometimes quite) different TCP variants perform similarly (i.e., with surprisingly little variation, given their technical differences) suggests that they are all hampered in a similar way. If we address this common issue, then we have the potential to improve circuit utilization across the board, for all of these TCP variants.



(a) Circuit utilization vs. TCP variant.



(b) Expected TCP seq. num. vs. time, for many TCP variants. Circuit days are shared in blue.

Figure 4: With 20 μs reconfiguration delays and 180 μs circuit days, no TCP variant performs well.

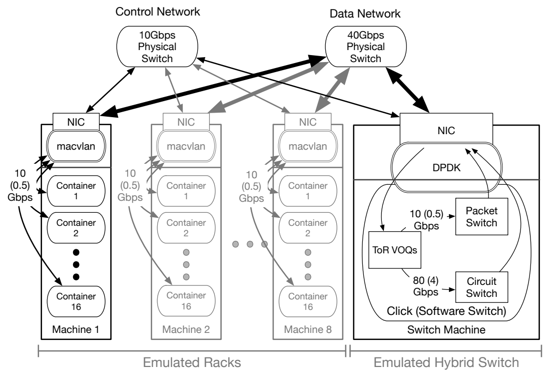


Figure 5: Etalon emulating 8 racks of 16 servers.

4 The Etalon Emulator

One of the key challenges in understanding TCP performance on RDCNs is performing repeatable experiments at scales appropriate for modern distributed cloud applications (i.e., across dozens or hundreds of hosts). In this section, we present our open-source emulator, Etalon [15], which measures the end-to-end performance of *real applications and end-host stacks* on emulated RDCNs in public testbeds.

4.1 Overview

Figure 5 presents an overview of Etalon. Each of the N physical machines emulates a rack of M servers using Docker containers [11, 44]. Containers are connected to the physical NIC using macvlan [12], which virtualizes a physical NIC into multiple virtual NICs, connecting them with a lightweight layer-2 software switch. `tc` [49] controls link bandwidths between the containers and the virtual switch, emulating a host-to-ToR link.

A separate physical machine emulates the reconfigurable datacenter network, as described in Section 4.2. Therefore, a cluster of $N+1$ physical machines can emulate $N \times M$ virtual hosts. For convenience, each physical host is connected to separate control and data networks, but this is not necessary. The experiment harness communicates with the testbed using RPyC [48]. Section 4.3 explains how time dilation enables Etalon to emulate many hosts with high-bandwidth links on a small testbed.

4.2 Click Software Switch

The $(N+1)^{\text{st}}$ machine emulates the RDCN itself, namely the ToR VOQs and the hybrid switch. This host runs a Click [34] software switch that uses DPDK [13] to process packets at line rate. We choose to emulate ToR VOQs in the software switch to make circuit and packet link emulation straightforward.

Figure 6 shows the software switch's internals. Packets enter the switch via DPDK [13] and are sent to an emulated ToR VOQ based on their *(source rack, destination rack)* pair. To achieve line rate, Etalon uses the Click elements `FromDPDKDevice` [7] and `ToDPDKDevice` [8] to exchange packets with the NIC. Packets are pulled from each VOQ by either the packet switch or the circuit switch. In Figure 6, packet uplink i is connected to the N VOQs in ToR i, pulling packets from these VOQs in a *round-robin fashion*. A packet pulled by a packet uplink enters the packet switch, where it is multiplexed over a packet downlink and transmitted using DPDK. If a packet would be dropped in the packet switch, it is held at the ToR VOQ (similar to PFC [28]). Circuit link i is connected to the ith VOQ of each of the N ToRs via a *pull switch*. A settable "input" value on pull switch i connects circuit link i to exactly one VOQ at a time. After packets traverse the circuit link, they are transmitted using DPDK. Before releasing a packet, the Click hybrid switch logs its IP and TCP headers, the current circuit state, and timing information. These logs are used offline to analyze the expected sequence number over time. The experiment harness communicates with the software switch using Click's control socket.

Our software switch contains three control elements (shown in gray in Figure 6): a demand estimator, a scheduler, and a schedule executor. The demand estimator estimates rack-to-rack demand using ToR VOQ occupancy. The scheduler computes a schedule from this demand, which is then run by the schedule executor by modifying the circuit link pull switches' "input" value (as described above). Our scheduler element is pluggable: We implement Solstice [39] as an example, but modify its objective to schedule maximal demand within a set window W (like Eclipse [2]), rather than scheduling all demand in unbounded time. We integrate Solstice, however, purely for implementation completeness. For our evaluation, a simple fixed strobe schedule, as described in

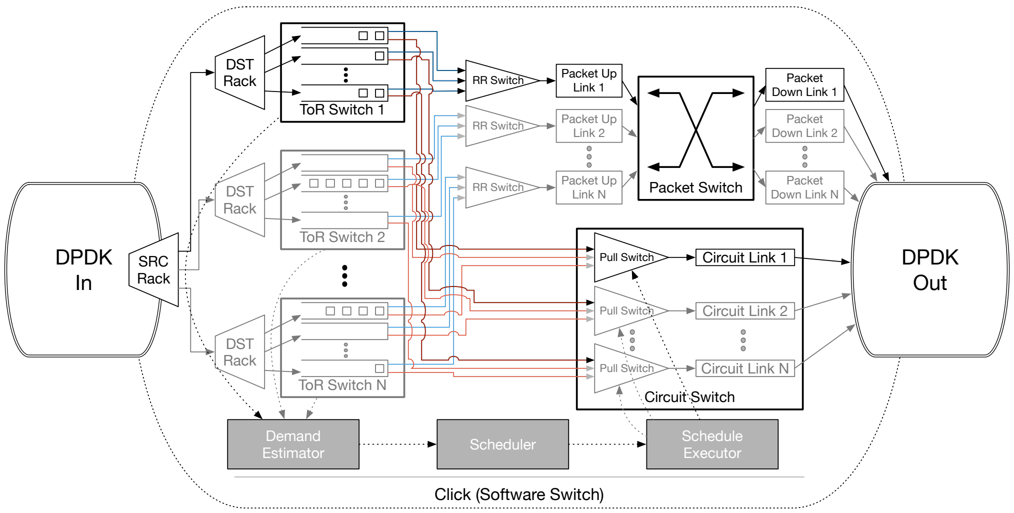


Figure 6: The Click software switch emulates the ToRs, the packet and circuit networks, and scheduling elements.

Section 4.5, is sufficiently illustrative.

4.3 Time Dilation

As the goal of Etalon is to emulate RDCNs on public testbeds, the machine emulating the hybrid network likely has only a single high-speed NIC. However, we wish to emulate a switch with N high-speed ports. We solve this problem with *time dilation (TD)*. Originally proposed for VMs [21, 22, 50] and recently containers [35, 55, 56], TD provides accurate emulation of higher-bandwidth links by "slowing down" the rest of the machine. We refer to the constant factor by which time is dilated as the *time dilation factor (TDF)*. We implement an open-source interposition library called LibVirtualTime (LibVT) [37], which applies TD to many common syscalls without requiring applications changes. We catch: `clock_gettime()`, `gettimeofday()`, `sleep()`, `usleep()`, `alarm()`, `select()`, `poll()`, and `setitimer()`. Extending LibVT to other syscalls is trivial. We verify that common network benchmarks (iperf [29], iperf3 [30], netperf [27], sockperf [41], flowgrind [58, 59], ping [46]) perform correctly with TD. We also limit CPU time for containers with respect to TD. Using time dilation to emulate high-speed links is one of Etalon's main advantages.

4.4 Etalon Testbeds

We use two testbeds for our experiments: a CloudLab cluster emulating 8 racks (128 hosts) and a local cluster emulating 3 racks (48 hosts). We use the large CloudLab cluster to validate the Etalon emulator, and run our experiments on the small local cluster. For the contributions in this paper, we do not require a large cluster or complex workload.

The local testbed uses four servers to emulate three racks of 16 machines plus the hybrid switch, as described in Section 4.2 and Figure 5. Each physical machine has 2 × 20-core 2.8 GHz Intel Xeon E5-2680v2 and is connected to a 40 Gb/s Ethernet data network (with jumbo frames). We also use Etalon on the public CloudLab APT cluster [9, 14], where nine R320 machines emulate eight racks of 16 hosts each and the hybrid switch. Each APT machine has an 8-core 2.1 GHz Intel Xeon E5-2450 and is connected to a 40 Gb/s Ethernet data network (with jumbo frames). At the time of our experiments, the CloudLab APT cluster was configured for 56 Gb/s InfiniBand, so we manually reconfigured the switches into 40 Gb/s Ethernet mode.

We use a TDF of 20× across our experiments. For example, in our 3-rack cluster, we emulate a 3-port 10 Gb/s (0.5 Gb/s)[4] packet switch and a 3-port 80 Gb/s (4 Gb/s) circuit switch. Outside of TD, the network can produce 3×0.5 Gb/s $+ 3 \times 4$ Gb/s = 13.5 Gb/s of total traffic, far below our data network's 40 Gb/s physical link speed. Each per-container link (i.e., each intra-rack link between the ToR and a host) is limited to 10 Gb/s (0.5 Gb/s).

Packet switch up/down links have 5 μs (100 μs with TDF) of delay each. Prior work assumes that the circuit network consists of ToRs connected to a pod-level central circuit switch, thus requiring long fibres (in the case of an optical network) [16]. To model this, we conservatively configure the circuit delay as 30 μs (600 μs), or 3× higher than the total packet link delay (5 $\mu s \times 2$ = 10 μs total). To avoid out-of-order packet delivery, if there is a circuit scheduled between racks S and D, then we disable the packet switch for rack pair (S, D) both during the reconfiguration leading up to a circuit and during the circuit period itself. Therefore, between S and D there exists exactly one connection: either packet or circuit. This is an example of non-segregated routing: Mice and elephant flows traverse the same links.

[4]Values in parenthesis represent bandwidth/delay outside of TD, i.e., actual bandwidth and delay values.

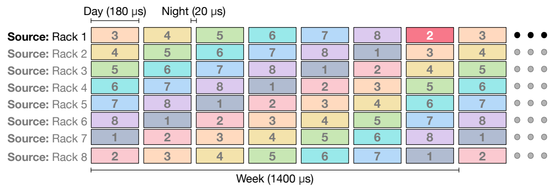

Day (180 μs)
Night (20 μs)
Source: Rack 1 3 4 5 6 7 8 2 3
Source: Rack 2 4 5 6 7 8 1 3 4
Source: Rack 3 5 6 7 8 1 2 4 5
Source: Rack 4 6 7 8 1 2 3 5 6
Source: Rack 5 7 8 1 2 3 4 6 7
Source: Rack 6 8 1 2 3 4 5 7 8
Source: Rack 7 1 2 3 4 5 6 8 1
Source: Rack 8 2 3 4 5 6 7 1 2
Week (1400 μs)

Figure 7: An 8-rack strobe schedule. All *(src rack, dst rack)* pairs can communicate 1/7th of the time.

4.5 Schedule and Workload

This paper focuses specifically on congestion control mechanisms, and therefore we believe that a simple traffic pattern is sufficient. With a couple of exceptions, we use the following schedule and workload for all experiments. Extending this investigation to complex workloads is future work.

Schedule We use a strobe schedule that, for a cluster with R racks, creates a circuit from a rack to each other rack $\frac{1}{R-1}$ of the time[5]. Figure 7 shows how, for an 8-rack schedule, a rack connects to each other rack in turn, repeating after a week. Solstice [39] would produce this schedule for an all-to-all workload such as a MapReduce shuffle [10]. For our experiments, a 25-rack cluster (described next) with 20 μs nights and 180 μs days (90% uptime) yields a week of duration $(20\ \mu s + 180\ \mu s) \times (25-1) = 4800\ \mu s$. The time between circuit days is $4800 - 180 = 4620\ \mu s$.

Workload The 16 emulated hosts on rack 1 each use flowgrind [58, 59] to send a TCP flow to their counterpart host on rack 2. The other racks are idle. We choose a flow duration of 3000 weeks, or 14.4 seconds for a 25-rack strobe schedule with 4800 μs per week. Each set of three weeks is treated as one experimental run, and the 1000 runs are averaged when reporting expected sequence number and VOQ length.

To better illustrate our contributions, we take advantage of our simple workload, described above, to emulate a larger testbed. Since our workload involves only racks 1 and 2, from the perspective of either of those racks, the cluster size effects only the duration between circuits. Therefore, we mimic a larger cluster by artificially lengthening this duration. We use this technique to run a 25-rack strobe schedule on our 3-rack local testbed.

4.6 Validating Etalon

We validate Etalon on the 8-rack CloudLab cluster using a strobe schedule while sending TCP traffic between pairs of racks for 2 seconds. ACKs are diverted around the switch for this one experiment to avoid ACK loss. By bypassing the hybrid switch, ACKs are transported instantly across the core of the emulated network, enticing TCP to ramp up faster, thus nullifying the bandwidth fluctuation described in Section 3. This is, of course, an unrealistic technique for actual networks, but we employ it to validate the Etalon emulator. We present timing and bandwidth results in Table 1. These results demonstrate that the emulator is sufficiently accurate to achieve the desired circuit schedule times (night and day lengths) and packet and circuit bandwidths.

[5] A rack need not connect to itself, so there are $R-1$ days in a week.

	Expected	Experimental Mean	Std. Dev.
Circuit day	180μs	180.25μs	0.04μs
Week length	1400μs	1400.02μs	0.05μs
Packet utilization	10 Gbps	9.93 Gbps	0.75 Gbps
Circuit utilization	80 Gbps	79.99 Gbps	1.60 Gbps

Table 1: Validating Etalon's timing and throughput.

5 Overcoming Rapid Bandwidth Fluctuation

As discussed in Section 3, the short uptimes (e.g., < 3 RTTs) of modern reconfigurable datacenter networks create bandwidth fluctuations such that TCP is unable to fully utilize the available bandwidth. This section describes two distinct techniques, implemented at different network layers, that adapt TCP to this challenge: 1) dynamic VOQ resizing that transparently prebuffers packets at the ToR before a circuit activates, and 2) explicit circuit state feedback to end-hosts that directly triggers a `cwnd` increase. Before introducing our techniques, however, we first consider a simpler, static-buffer approach that illustrates the bandwidth and latency trade-off that our solutions must navigate.

Section 3.2 presented the general signals that TCP variants use to infer network congestion. Increasing switch buffer sizes masks packet loses caused by queue overflows. For loss-based TCP variants like CUBIC and New Reno, removing this congestion signal triggers ramp-up. However, this technique is not appropriate for delay-based variants such as TIMELY or Vegas. We demonstrate in Section 6.2 that for variants which rely at least partially on loss as a congestion signal, hiding congestion-based drops is effective at increasing circuit utilization. Our second technique, explicit circuit state feedback via the `ECE-echo` bit, is more general since it provides a direct signal to end-hosts, but of course has the trade-off that it requires modifying end-hosts.

5.1 Leveraging VOQs to Increase Bandwidth

It is well understood that switch buffer sizing presents a trade-off between high bandwidth and low latency [5, 17]. The more traffic that can be queued up in the network, the better it will be able to saturate links in the presence of transient demand—or, in our case, capacity variations—because packets will be available to burst immediately in the event of a capacity increase. However, when packets incur queuing latency, the effective round-trip time increases. Because latency (and tail latency in particular) impacts short flow—and potentially job—completion times, much work on datacenter transport protocols has focused on achieving high bandwidth while keeping queues short [1,5]. As a starting point toward mitigating the effects of bandwidth fluctuation in RDCNs, we experiment with various sizes of static ToR VOQs and demonstrate that loading the network with excess traffic does help saturate the high-bandwidth circuits, but, as expected, at the cost of high latency.

Using the schedule and workload described in Section 4.5, we configure the hosts to run TCP CUBIC, vary the size of the ToR VOQs from 4 to 128 packets, and examine the impact on circuit utilization. Figure 8a. shows utilization measured as the aggregate achieved bandwidth of all of a rack's flows versus the maximum bandwidth the flows should have been able to achieve, averaged over all of the circuit periods in an experiment. For small buffers, circuit utilization is low. Large buffers fare better, with 64 packets building up a sufficient "backlog" of packets to absorb the bandwidth fluctuations.

Figure 8b shows the expected TCP sequence number during the lead-up to a circuit day, for various queue sizes, as measured by the software switch (Section 4.2). The slope of each line is a flow's achieved bandwidth. The *optimal* and *packet only* baselines are computed as in Figure 2. Larger buffers yield a steady convergence to *optimal*. While TCP grows at a rate of one packet per RTT, regardless of buffer size, larger switch buffers allow flows to queue up a packet backlog which then drains during circuit uptime, as shown by the dotted VOQ lines. The VOQ length is level throughout the packet network period.

Finding the "proper" VOQ size for a hybrid switch is difficult. Common wisdom is to use the bandwidth-delay product (BDP) of the network, but the BDP is different for the packet network and the circuit network: ~2 and ~34 packets, respectively. A time-weighted average based on the schedule suggests ~8 packets may be appropriate. As shown in Figure 8a, however, none of these values provide full circuit utilization: 64+ packets are needed[6]. However, we cannot simply adopt queues this large because of their high latency. In a datacenter, where link lengths are short, queuing delay impacts tail latency, which in turn directly impacts distributed applications [1]. Figure 8c shows 99th percentile tail latency for the various ToR VOQ sizes, measured as each packet enters and leaves the software switch. As expected, the packet and circuit latencies both grow as we increase the buffer size.

We want the best of both worlds: Can we achieve full circuit utilization while simultaneously not incurring a latency penalty? No static buffer configuration achieves this. The following subsections describe two techniques to meet this goal: *dynamic* buffer resizing (Section 5.2) and explicit circuit state feedback to end-hosts (Section 5.3).

[6]Our later experiments show that ~45 packets may suffice.

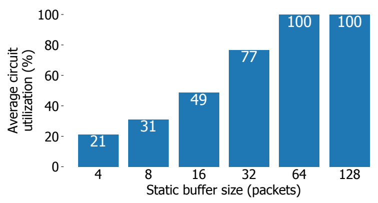
5.2 Dynamic Buffer Resizing

We propose dynamically resizing ToR VOQ capacity to remedy the effects of rapid bandwidth fluctuations on TCP. This is an entirely *in-network* solution, which, to our knowledge, has not been explored in the context of network scheduling. In this section, we focus specifically on loss-based variants of TCP, such as CUBIC.

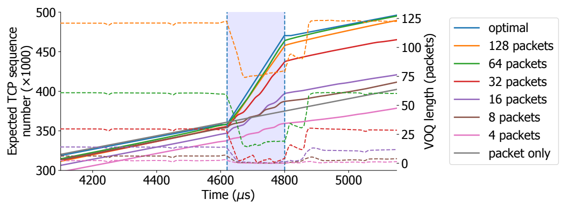
Our key insight is that bandwidth fluctuation within RDCNs is not arbitrary: It is part of a schedule and is known in advance. With this knowledge, we can align in-network buffer sizes with either the packet switch or the circuit switch in real time. By itself, a packet switch can effectively achieve full throughput with a very small buffer (e.g., 4 packets), whereas large buffers cause queuing delay, as shown in Figure 8c. The previous section demonstrated that a circuit switch needs larger buffers (e.g., 64+ packets) to achieve full utilization due to bandwidth fluctuations. With dynamic buffer resizing, we take a step in the right direction by keeping buffers shallow when the packet switch is in use and deep when a circuit is active. Doing this naïvely (i.e., resize buffers when the circuit comes up) provides little benefit; there is simply not enough time in one day for TCP to grow to fill the circuit link, regardless of how large the buffer. Data needs to be available *immediately* at circuit start (either buffered or via a high TCP sending rate); ramping up *ex post facto* means that circuit time has already been wasted.

Instead, we dynamically resize ToR VOQs for a *(source, destination)* rack pair *in advance* of a circuit starting for that pair. This implies that if the circuit schedule dictates that this rack pair should spend most of the time using the packet switch, then small buffers will be used to avoid incurring additional latency. Increasing a rack pair's VOQ size provides two benefits: 1) Packets build up in the queue and are then drained immediately when the circuit activates, creating a momentary burst of traffic, and 2) When the buffered packets drain at circuit start, they generate a surge of ACKs that increase the sender's congestion window (`cwnd`) and sending rate. Exactly how quickly `cwnd` grows depends on the TCP variant in use.

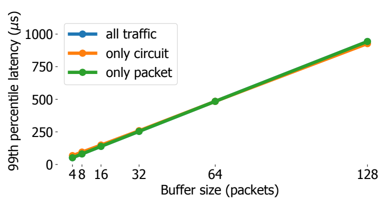
Our buffer resize function has three parameters: `resize`(s, b, τ), where s and b are the small and large buffer sizes in packets, respectively, and τ is how early a buffer should be resized in advance of the circuit start. For the rest of the paper, we use $s = 16$ and $b = 50$. τ is a trade-off: Resizing too late means low circuit utilization, but resizing too early increases latency. We vary τ in experiments in Section 6.1. While the value of τ impacts the circuit utilization/latency trade-off, we find that waiting to resize the buffer back to s after a circuit stops (and thus retaining large buffers for a short time while the packet network comes back into use) has no benefit. Thus, we always reset buffers back to s immediately after circuit teardown.



(a) Circuit utilization vs. static buffer size.



(b) Expected TCP sequence number vs. time, for various static buffer sizes. Circuit days are shaded in blue. Dotted lines are the corresponding VOQ length.



(c) 99th percentile tail latency vs. static buffer size.

Figure 8: Tradeoffs of various static VOQ sizes. Larger buffers improve utilization at the cost of latency.

5.3 Incorporating Explicit Network Feedback

Dynamically resizing ToR VOQs is a transparent, in-network technique that raises circuit utilization without harming latency and does not require end-host modifications. However, this approach is intended for loss-based congestion control schemes and does entail additional latency and buffering. In this section, we propose a direct form of circuit state feedback that applies to more TCP variants (e.g., delay and explicit-feedback–based schemes) and ultimately mitigates this latency penalty. Of course, the trade-off is that techniques involving explicit feedback require modifying the TCP congestion control algorithm running at the sender, making them non-transparent and more difficult to deploy.

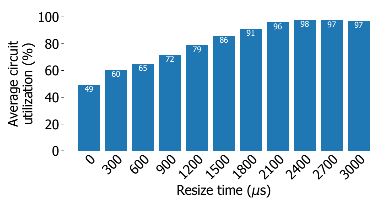
The idea behind explicit network feedback is simple: Notify the sender when a flow is traversing a circuit and should ramp up. However, the delay in this feedback reaching the sender is crucial. Since circuits are live for as few as 3 RTTs, a signal that requires 1 RTT to propagate to the sender (e.g., marking ECN bits of outgoing flows), provides limited benefit. We tighten the feedback loop by instead marking ACKs as they return to the sender. For some rack pair (S,D), we modify our software switch to set the `ECN-echo` (ECE) bit in the TCP headers of ACKs sent by D, if there is currently a circuit enabled from S to D. The hybrid switch examines the circuit state and marks ACKs *after* they traverse the circuit link, so the "freshness" of the feedback signal is equal to the propagation delay of a single packet link between the ToR and the sending end-host, S.

We create a pluggable TCP congestion control module for Linux called reTCP (REconfigurable datacenter network TCP) which looks at this stream of ECE bits, multiplicatively increasing its `cwnd` by $\alpha \geq 1$ on $0 \rightarrow 1$ transitions and decreasing it by $0 \leq \beta \leq 1$ on $1 \rightarrow 0$ transitions. reTCP is an edge detector: It modifies `cwnd` on ECE-bit state *transitions*, not for every packet. We set $\alpha = 2$ and $\beta = 0.5$, based on the results of a parameter sweep. Intuitively, this provides higher circuit utilization because TCP will immediately have a higher sending rate when a circuit starts.

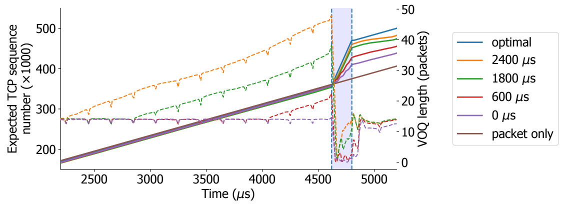
Our reTCP implementation is based on TCP New Reno [23] and relies on its congestion control algorithm, in addition to the above-mentioned technique. reTCP also requires a single-line kernel change, as the kernel only passes ECE flags to congestion control modules if ECN is enabled. Enabling ECN shrinks `cwnd` upon receiving an ECE-marked packet (because ECE bits typically convey congestion information), so we leave ECN disabled and instead modify the kernel to always pass the ECE flag to reTCP.

reTCP is beneficial on its own by increasing the TCP sending rate when a circuit begins, but in combination with dynamic buffer resizing, its benefits multiply. As a starting point, consider the period when the packet network is active, operating with static VOQs (not dynamic buffer resizing). The sender ramps up to a steady state that saturates the packet network bandwidth and VOQ capacity. Suppose that we trigger a reTCP `cwnd` increase during this regime. The network would drop the additional packets and the sender would slow back down, with no benefit. However, dynamic buffer resizing is specifically designed to provide extra VOQ capacity to flows before circuit start, but the usefulness of this capacity and for how long it must be available is determined by the sender's ramp-up rate. The two techniques play to each other's strengths: Combining them, dynamic buffer resizing provides more capacity for flows, and then reTCP fills it. When used in this way, we modify reTCP to mark ACKs not at the start of a circuit, but at the start of the dynamic buffer resizing period. By increasing `cwnd` at the same time that the VOQs grow, reTCP quickly fills the larger capacity, dramatically reducing the duration of prebuffering required to achieve full utilization. Bringing the collaboration full-circle, reTCP's ramp-up burst reduces the prebuffering duration, which in turn lessens the time fraction during which the network experiences deep buffers, thus mitigating the ensuing tail latency spike. Section 6.3 visualizes how dynamic buffer resizing and reTCP jump-start the TCP sender just in time for the circuit network.

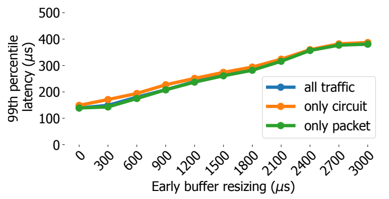
Instead of communicating circuit state on-demand via ACKs and their ECE bits, an alternative design is to inform end-hosts directly using control plane messages, similar to how the ToRs are notified to increase their VOQ capacity prior to circuit start. This would allow the sender to increase its `cwnd` exactly when desired, instead of after the first marked ACK arrives. However, this requires invasive modifications to the end-hosts, since the sender TCP stack would need to communicate with the central scheduler. Our approach is



(a) Circuit utilization vs. prebuffering duration.



(b) Expected TCP sequence number vs. time, for various prebuffering durations. Circuit days are shaded in blue. Dotted lines are the corresponding VOQ length.



(c) 99th percentile latency vs. prebuffering duration.

Figure 9: Dynamic buffer resizing improves circuit utilization at the expense of tail latency.

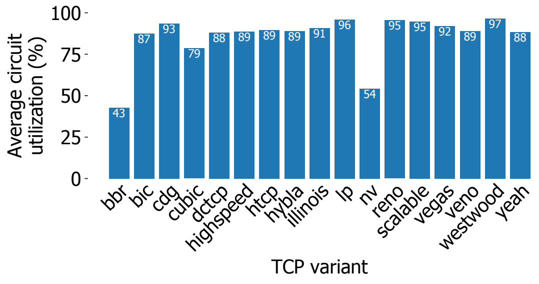


Figure 10: Circuit utilization vs. TCP variant.

less reactive but more practical because it uses existing TCP header fields, modifies only a few lines of code, and avoids distributed control challenges like time synchronization.

6 Evaluation

This section demonstrates that our two proposed techniques, dynamic buffer resizing and explicit circuit state feedback, overcome the challenge of bandwidth fluctuation in RDCNs and enable TCP to take advantage of high-bandwidth circuits when they become available. Additionally, we demonstrate that dynamic buffer resizing provides a general benefit to many TCP variants (beyond CUBIC).

6.1 Evaluating Dynamic Buffer Resizing

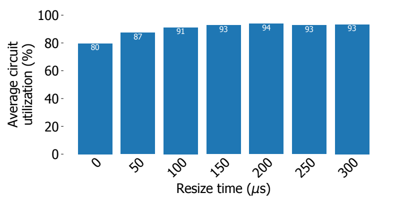
To test the efficacy of dynamic in-network buffer resizing, we repeat the experiments from Section 5.1, but with dynamic instead of static ToR VOQs. Using the schedule and workload described in Section 4.5, we configure the hosts to run TCP CUBIC and vary how early buffer resizing takes place, τ, from 0 μs to 3000 μs, in intervals of 300 μs. Buffers switch between a short capacity of 16 packets and a large capacity of 50 packets. Figure 9a shows average circuit utilization for the various τ. The earlier we resize, the higher utilization flows achieve. With $\tau = 1800$ μs, circuit utilization increases by 1.87× (48.6% → 91.1%), compared to 16-packet static queues.

Figure 9b shows a graph of the expected TCP sequence number and the VOQ length during the lead-up to a circuit day, for selected τ. VOQ length hovers at the small buffer size until dynamic buffering takes effect, then grows steadily, and finally drains sharply when the higher bandwidth circuit activates. For situations that see high utilization, sufficiently many ACKs return and ramp up the sending rate before the VOQs drain completely, which, for TCP CUBIC, is effectively achieved at $\tau = 1800$ μs.

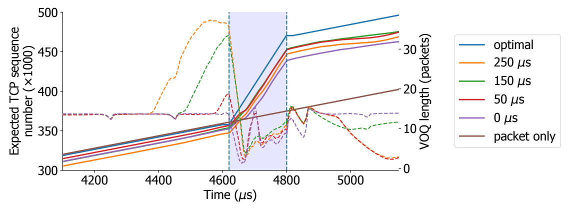
The queue length required for high utilization is a function of the BDPs of the packet (10 Gb/s × 2 × 5 μs = 100 Kb) and circuit (80 Gb/s × 30 μs = 2.4 Mb) networks. These BDPs differ by ~32 9000 B packets. Assuming a few extra packets due to store-and-forward delays, between 35 and 40 packets must be in the VOQs before circuit start to keep the network fully utilized. The matches the growth of the VOQ line for $\tau = 1800$ μs (91% utilization) in Figure 9b.

The remaining question is whether this high utilization comes at the cost of high latency, as it did for static buffers. Median latency does not increase until $\tau = 2700$ μs; we omit the results for brevity. Figure 9c shows how dynamic resizing affects 99th percentile tail latency. Tail latency depends on the length of the VOQs, which increases with the prebuffering duration. The 1.87× circuit utilization increase is paired with a tail latency growth of 2.33× (123 μs → 286 μs). These results can be compared to static VOQs in two ways: (1) Comparing $\tau = 1800$ μs to a static buffer with similar throughput (64 packets), dynamic buffering improves tail latency by 0.59× (484 μs → 286 μs); (2) Comparing to a static buffer with similar tail latency (32 packets), the circuit utilization increases by 1.19× (76.7% → 91.1%). Ultimately, this experiment demonstrates that dynamic buffer resizing has the potential to meet our goal of achieving full circuit utilization with less of an impact on latency than large static buffers, but its latency penalty, especially at the tail, is still unreasonably high for distributed applications.

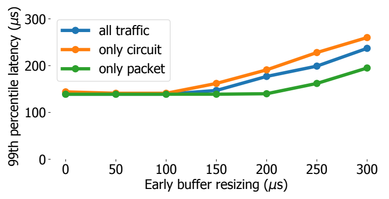
Given that resizing provides large buffers to flows for a significant amount of time (e.g., 41% of the schedule for $\tau = 1800$ μs) it is surprising that tail latency is not as bad as static buffers for comparable circuit utilization. We can use large buffers for a large fraction of time because when circuits are torn down and flows transition back to the packet network, the resulting bandwidth reduction (8× in these experiments) causes TCP to dramatically scale back its sending rate at



(a) Circuit utilization vs. prebuffering duration.



(b) Expected TCP sequence number vs. time, for various prebuffering durations. Circuit days are shaded in blue. Dotted lines are the corresponding VOQ length.



(c) 99th percentile latency vs. prebuffering duration.

Figure 11: reTCP achieves high utilization with much lower tail latency than dynamic buffers alone.

the same moment when the VOQs shrink. In effect, switching back to the packet network resets TCP and the networks experiences a period of short queues while TCP recovers.

However, a 2.33× tail latency increase is unacceptable for many latency-sensitive applications. We have not yet met our goal of achieving the "best of both worlds" of high throughput without the corresponding latency penalty. Section 6.3 demonstrates how reTCP completes the picture, achieving high utilization with only a 1.20× tail latency increase.

6.2 Benefits for all TCP Variants

In Section 3.2.4, we demonstrated that the problem of bandwidth fluctuation impacts many TCP variants we tested. Returning to the experiment in Figure 4, we evaluate the variants' performance with dynamic buffer resizing, with $\tau = 1200$ μs. Figure 10 shows the utilization achieved by the 17 variants. The variants experience an average utilization improvement of 36%, compared to static 16-packet buffers, without any modifications to the end-hosts. Resizing earlier (e.g., $\tau = 1800$ μs) yields still-higher utilization, but we present $\tau = 1200$ μs to better illustrate how the TCP variants respond differently. BBR [5] and TCP NV [3], being delay-based protocols that seek to keep buffer occupancy low, naturally do not take advantage of dynamic buffer resizing and realize only 43% and 54% utilization, respectively. Note that DCTCP [1] relies on explicit congestion feedback from network switches in the form of a stream of ECN marks set if adding a packet to a switch queue would cause the queue's capacity to pass a threshold. To accurately support DCTCP in the Etalon emulator, we modify the hybrid switch to mark packets in this way, at 10 packets when using small queues (16 packets) and 31 packets when using large queues (50 packets).

6.3 Evaluating Explicit Network Feedback

Conveying circuit state information to end-hosts provides an explicit signal that TCP can use to adapt to bandwidth fluctuations in RDCNs. In isolation, reTCP yields higher average circuit utilization than static buffers alone, with an average improvement of 2.65% across the static buffer capacities in Figure 8a. An improvement of 6% with 32-packet buffers is the most significant. For 64 and 128–packet queues, both static buffers and reTCP achieve full utilization. We omit the graphs for brevity.

When used in combination with dynamic buffer resizing, reTCP achieves high circuit utilization with a shorter prebuffering duration and, in turn, lower tail latency. Figure 11 repeats the experiments from Section 6.1, but with both dynamic buffer resizing and reTCP, and instead varies τ from 0 μs to 300 μs, in intervals of 50 μs. Note that the x-axis range in Figure 11b is different than in Figure 9b to better visualize the range of τ values during which reTCP ramps up. Figure 11a shows that the two techniques working together achieve a 1.91× (48.7% → 92.7%) circuit utilization improvement compared to 16-packet static buffers, but with lower τ than dynamic buffers alone: at $\tau = 150$ μs instead of $\tau = 1800$ μs. The impact of this order-of-magnitude decrease in prebuffering duration manifests itself in lower tail latency in Figure 11c: an only 1.20× increase (123 μs → 147 μs), compared to 2.33× for dynamic buffer resizing alone.

Comparing Figures 9b and 11b, the steady VOQ growth is replaced by a jump as `cwnd` doubles, sending more packets into the network. At $\tau = 150$ μs, the sender injects sufficiently many packets to grow the VOQ large enough to saturate the higher BDP of the circuit network, when it becomes active. The rate of growth is quick: It keeps the required prebuffering duration short, which reduces tail latency. Overall, dynamic buffer resizing and reTCP overcome the challenge of bandwidth fluctuation in RDCNs, increasing circuit utilization by 1.91× with an only 1.20× tail latency penalty.

6.4 Limitations

Increasing the VOQ size does not, on its own, lead to higher circuit utilization. The TCP sender must ramp up to fill the additional capacity. Section 6.1 shows that this ramp-up may take milliseconds, posing a challenge for schedules that allocate circuits between some rack pairs frequently: The circuit downtime period may be too short to support this lengthy ramp-up. E.g., the time between circuits for a 3-rack cluster with a strobe schedule (week length = 400 μs) is 220 μs, far lower than the $\tau = 1800$ μs deemed necessary by Section 6.1. Investigating this case revealed that as the prebuffer start time approaches the end of the previous circuit, the residual high sending rate quickly fills the VOQs, causing the network to enter a mode similar to using large static buffers, with high tail latency. reTCP ameliorates this problem.

7 Related Work

Research into RDCN design [6,16,20,25,26,32,38,47,51,57] and scheduling [2,36,39] has yet to examine TCP-related challenges for modern RDCNs. c-Through [51] proposes resizing end-host network buffers, but for the purpose of traffic batching, which is necessary to ensure that senders have enough data to fill a circuit when it becomes available. This is only an issue for circuits with long (e.g., millisecond-scale) uptimes, not the microsecond-scale technologies we address in this paper. Other work avoids the bandwidth fluctuation problem by segregating traffic to use either the packet or circuit networks exclusively [16, 38]. Our techniques simplify the network design by not segregating traffic.

Many of the TCP variants in Section 3.2 are tailored for high-bandwidth networks, but they assume that the bandwidth does not change on short timescales. Prior work [52] has examined how TCP reacts to bandwidth fluctuations in wireless networks, but these networks are fundamentally different than RDCNs. Wireless networks have lower bandwidths and BDPs than RDCNs, and unlike in wireless networks, bandwidth fluctuations in RDCNs are not random: They are part of a schedule that is known in advance. This insight enables proactive techniques like dynamic VOQ resizing.

8 Conclusion

With new advances in RDCN technology comes the need to reexamine the protocols running on top of these networks. This paper proposes two techniques to adapt TCP to the rapid bandwidth fluctuation inherent in microsecond-scale reconfigurable datacenter networks: 1) In the network, we transparently resize ToR VOQ buffers prior to circuit activation to help TCP ramp up, but at the cost of higher tail latency; 2) Involving the end-hosts opens the door for high utilization without a latency penalty by incorporating explicit circuit state signals sent from the hybrid switch. Etalon provides opportunities for future work, e.g., exploring multicast-enabled optical circuit switching (e.g., Blast [53]), providing a cross-cutting evaluation of different RDCN designs, and investigating challenges in future sub-μs RDCNs. While this paper focuses on TCP specifically, our investigation into implicit and explicit techniques for adapting the sending rate to the state of the network applies to congestion control in general. We believe that our experiences speak to the need for an end-to-end evaluation of future RDCN designs.

Acknowledgements We would like to thank the anonymous reviewers and our shepherd, Manya Ghobadi, for their helpful feedback. This work was supported in part by NSF grant CNS-1565343.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proc. ACM SIGCOMM*, August 2010.

[2] Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. Costly circuits, submodular schedules and approximate carathéodory theorems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 44, pages 75–88. ACM, 2016.

[3] Lawrence Brakmo. Tcp-nv: Congestion avoidance for data centers. Linux Plumbers Conference 2010, 2010.

[4] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. *SIGCOMM Comput. Commun. Rev.*, 24(4):24–35, October 1994.

[5] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[6] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, and Xitao Wen. OSA: An Optical Switching Architecture for Data Center Networks and Unprecedented Flexibility. In *Proc. USENIX NSDI*, April 2012.

[7] Click. FromDPDKDevice element documentation. https://github.com/kohler/click/wiki/FromDPDKDevice, 2020.

[8] Click. ToDPDKDevice element documentation. https://github.com/kohler/click/wiki/ToDPDKDevice, 2020.

[9] CloudLab. CloudLab. https://www.cloudlab.us/, 2018.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Docker. Docker. https://www.docker.com/, 2018.

[12] Docker. Get started with Macvlan network driver. https://docs.docker.com/engine/userguide/networking/get-started-macvlan/, 2018.

[13] DPDK. DPDK. https://dpdk.org, 2018.

[14] Emulab. APT cluster. https://www.aptlab.net/, 2018.

[15] Etalon. Etalon: A reconfigurable datacenter network (rdcn) emulator. https://github.com/mukerjee/etalon, 2020.

[16] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proc. ACM SIGCOMM*, August 2010.

[17] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(04):397–413, jul 1993.

[18] Sally Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649 https://rfc-editor.org/rfc/rfc3649.txt, December 2003.

[19] Klaus-Tycho Foerster, Manya Ghobadi, and Stefan Schmid. Characterizing the algorithmic complexity of reconfigurable data center architectures. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, page 89–96, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 216–229. ACM, 2016.

[21] Diwaker Gupta, Kashi V. Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex C. Snoeren, and Geoffrey M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems (TOCS)*, 29(2):4:1–4:48, May 2011.

[22] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C Snoeren, Amin Vahdat, and Geoffrey M Voelker. To infinity and beyond: time warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2. ACM, 2005.

[23] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582 https://rfc-editor.org/rfc/rfc6582.txt, April 2012.

[24] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[25] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting Data Center Networks with Multi-gigabit Wireless Links. In *Proc. ACM SIGCOMM*, August 2011.

[26] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 319–330, New York, NY, USA, 2014. ACM.

[27] HewlettPackard. netperf. https://github.com/HewlettPackard/netperf, 2018.

[28] IEEE. 802.1Qbb – Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/, 2018.

[29] iperf. iperf. https://iperf.fr/, 2018.

[30] iperf3. iperf3. https://iperf.fr/, 2018.

[31] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, August 1988.

[32] Srikanth Kandula, Jitendra Padhye, and Paramvir Bahl. Flyways To De-Congest Data Center Networks. In *Proc. ACM HotNets-VIII*, October 2009.

[33] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, page 89–102, New York, NY, USA, 2002. Association for Computing Machinery.

[34] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[35] Jereme Lamps, David M Nicol, and Matthew Caesar. Timekeeper: A lightweight virtual time system for linux. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 179–186. ACM, 2014.

[36] Conglong Li, Matthew K Mukerjee, David G Andersen, Srinivasan Seshan, Michael Kaminsky, George Porter, and Alex C Snoeren. Using indirect routing to recover from network traffic scheduling estimation error. In

Proceedings of the Symposium on Architectures for Networking and Communications Systems, pages 13–24. IEEE Press, 2017.

[37] libVT. libVT: user space virtual time library. https://github.com/mukerjee/libvt, 2020.

[38] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit Switching Under the Radar with REACToR. In *Proc. USENIX NSDI*, April 2014.

[39] He Liu, Matthew K Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M Voelker, David G Andersen, Michael Kaminsky, et al. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 41. ACM, 2015.

[40] Shao Liu, Tamer Baundefinedar, and R. Srikant. Tcp-illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools*, valuetools '06, page 55–es, New York, NY, USA, 2006. Association for Computing Machinery.

[41] Mellanox. sockperf. https://github.com/Mellanox/sockperf, 2018.

[42] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. RotorNet: a scalable, low-complexity, optical datacenter network. In *Proc. ACM SIGCOMM*, August 2017.

[43] William M Mellette, Alex C Snoeren, and George Porter. P-fattree: A multi-channel datacenter network topology. In *HotNets*, pages 78–84, 2016.

[44] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[45] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of ACM SIGCOMM 2015*, SIGCOMM '15. ACM, 2015.

[46] ping. ping(8) - linux man page. https://linux.die.net/man/8/ping, 2020.

[47] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang-Chen Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating Microsecond Circuit Switching into the Data Center. In *Proc. ACM SIGCOMM*, August 2013.

[48] RPyC. RPyC - transparent, symmetric distributed computing. https://rpyc.readthedocs.io/en/latest/, 2020.

[49] tc. tc(8) - linux man page. https://linux.die.net/man/8/tc, 2020.

[50] Kashi Venkatesh Vishwanath, Diwaker Gupta, Amin Vahdat, and Ken Yocum. Modelnet: Towards a datacenter emulation environment. In *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*, pages 81–82. IEEE, 2009.

[51] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T. S. Eugene Ng, Michael Kozuch, and Michael Ryan. c-Through: Part-time Optics in Data Centers. In *Proc. ACM SIGCOMM*, August 2010.

[52] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, NSDI '13. USENIX, 2013.

[53] Yiting Xia, TS Eugene Ng, and Xiaoye Steven Sun. Blast: Accelerating high-performance data analytics applications by optical multicast. In *Proc. IEEE INFOCOM*, pages 1930–1938. IEEE, 2015.

[54] Lisong Xu, K. Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *Proceedings of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2514 – 2524 vol.4. IEEE, 04 2004.

[55] Jiaqi Yan and Dong Jin. A virtual time system for linux-container-based emulation of software-defined networks. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 235–246. ACM, 2015.

[56] Jiaqi Yan and Dong Jin. Vt-mininet: Virtual-time-enabled mininet for scalable and accurate software-define network emulation. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 27. ACM, 2015.

[57] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y. Zhao, and Haitao Zheng. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *Proc. ACM SIGCOMM*, August 2012.

[58] Alexander Zimmermann, Arnd Hannemann, and Tim Kosse. Flowgrind-a new performance measurement tool. In *IEEE GLOBECOM 2010*, pages 1–6. IEEE, 2010.

[59] Alexander Zimmermann, Arnd Hannemann, and Tim Kosse. flowgrind. http://www.flowgrind.net/, 2018.

A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency

Tom Barbette Chen Tang Haoran Yao Dejan Kostić
Gerald Q. Maguire Jr. Panagiotis Papadimitratos Marco Chiesa
KTH Royal Institute of Technology

Abstract

Large service providers use load balancers to dispatch millions of incoming connections per second towards thousands of servers. There are two basic yet critical requirements for a load balancer: *uniform load distribution* of the incoming connections across the servers and *per-connection-consistency* (PCC), *i.e.*, the ability to map packets belonging to the same connection to the same server even in the presence of changes in the number of active servers and load balancers. Yet, meeting both these requirements at the same time has been an elusive goal. Today's load balancers minimize PCC violations at the price of non-uniform load distribution.

This paper presents CHEETAH, a load balancer that supports uniform load distribution and PCC while being scalable, memory efficient, resilient to clogging attacks, and fast at processing packets. The CHEETAH LB design guarantees PCC for *any* realizable server selection load balancing mechanism and can be deployed in both a stateless and stateful manner, depending on the operational needs. We implemented CHEETAH on both a software and a Tofino-based hardware switch. Our evaluation shows that a stateless version of CHEETAH guarantees PCC, has negligible packet processing overheads, and can support load balancing mechanisms that reduce the flow completion time by a factor of $2-3$x.

1 Introduction

The vast majority of services deployed in a datacenter need load balancers to spread the incoming connection requests over the set of servers running these services. As almost half of the traffic in a datacenter must be handled by a load balancer [41], the inability to uniformly distribute connections across servers has expensive consequences for datacenter and service operators. The most common yet cost-ineffective way of dealing with imbalances and meet stringent Service-Level-Agreements (SLAs) is to over-provision [13].

Existing LBs rely on a simple hash computation of the connection identifier to distribute the incoming traffic among the servers [3, 13, 15, 20, 37, 41, 53]. Recent measurements on Google's production traffic showed that hash-based load balancers may suffer from load imbalances up to 30% [13].

A natural question to ask is why existing load balancers do not rely on more sophisticated load balancing mechanisms, *e.g.*, weighted round robin [51],"power of two choices" [33], or least loaded server. The answer lies in the extreme dynamicity of cloud environments. Services and load balancers "*must be designed to gracefully withstand traffic surges of hundreds of times their usual loads, as well as DDoS attacks*" [3]. This means that the number of servers and load balancers used to provide a service can quickly change over time. Guaranteeing that packets belonging to existing connections are routed to the correct server despite dynamic reconfigurations requires *per-connection-consistency* (PCC) [32] and has been the focus of many previous works [3, 13, 15, 20, 32, 37, 41]. When only the number of load balancers change, hash-based load balancing mechanisms guarantee PCC as packets reach the correct server even when hitting a different LB [3, 37]. To deal with changes in the numbers of servers, existing LBs either store the "connection-to-server" mapping [13, 20, 32, 41] or let the servers reroute packets that were misrouted [3, 37]. In both cases, a hash function helps mitigate PCC violations, though it cannot completely avoid them (more details in Sect. 2). To summarize, existing load balancers cannot uniformly distribute connections across the servers as they rely on hash functions to mitigate (but not avoid) PCC violations.

This paper presents the design and evaluation of CHEETAH, a load balancer (LB) system with the following properties:

- ***dynamicity***, the number of LBs and servers can increase or decrease depending on the actual load;
- ***per-connection-consistency*** (PCC), packets belonging to the same connection are forwarded to the same server;
- ***uniform load distribution***, by supporting advanced load balancing mechanisms that efficiently utilize the servers;
- ***efficient packet processing***, the LB should have minimal impact on communication latency; and
- ***resilience***, it should be hard for a client to "clog" the LB

and the servers with spurious traffic.

CHEETAH takes a different approach compared to existing LBs. CHEETAH stores information about the connection mappings into the connections themselves. More specifically, when a CHEETAH LB receives the first packet of a connection, it encodes the selected server's identifier into a *cookie* that is permanently added to all the packet headers exchanged within this connection. Unlike previous work, which relies on hash computations to mitigate PCC violations, the design of CHEETAH completely *decouples* the load balancing logic from PCC support. This in turn allows an operator to guarantee PCC regardless of the "connection-to-server" mapping produced by the chosen load balancing logic. The goal of this paper is not the design of a novel load balancing mechanism for uniformly spreading the load but rather the design of CHEETAH as a building block to support PCC for *any* realizable load balancing mechanisms *without* violating PCC. As for resilience, we cannot expose the server identifiers to users as this would open the doors to clogging a targeted server. CHEETAH is designed with resilience in mind, thwarting resource exhaustion and selective targeting of servers. To this end, CHEETAH generates "opaque" cookies that can be processed fast and can only be interpreted by the LB.

We present two different implementations of CHEETAH, a stateless and a stateful version. Our stateless and stateful CHEETAH LBs carefully encode the connection-to-servers mappings into the packet headers so as to guarantee levels of resilience that are no worse (and in some cases even stronger) than existing stateless and stateful LBs, respectively. For instance, our stateful LB increases resilience by utilizing a novel and fast stack-based mechanism that dramatically simplifies the operation of today's cuckoo-hash-based stateful LBs, which suffer from slow insertion times.

In summary, our contributions are:

- We quantify limitations of existing stateless and stateful LBs through large-scale simulations. We show that the quality of the load distribution of existing LBs is 40 times worse than that of an ideal LB. We also show stateless LBs (such as Beamer and Faild) can reduce such imbalances at the price of increasing PCC violations.
- We introduce CHEETAH, an LB that guarantees PCC for any realizable load balancing mechanisms. We present a stateless and a stateful design of CHEETAH, which strike different trade-offs in terms of resilience and performance.
- We implement our stateless and stateful CHEETAH LBs in FastClick [5] and compare their performance with state-of-the-art stateless and stateful LBs, respectively. We also implement both versions of CHEETAH with a weighted round-robin LB on a Tofino-based switch [6].
- In our experiments, we show the potential benefits of CHEETAH with a non-hash-based load balancing mechanism. The number of processor cycles per packet for *both* our stateless and stateful implementation of CHEETAH is comparable to existing stateless implementations and 3.5x less than existing stateful LBs.

2 Background and Motivation

Internet organizations deploy large-scale applications using clusters of servers located within one or more datacenters (DCs). We provide a brief background on DC load balancers, discuss related work, and show limitations of the existing schemes. We do not discuss geo-distributed load balancing across DCs. Further, we distinguish between *stateless* LBs, which do not store any per-connection state, and *stateful* LBs, which store some information about ongoing connections.

Multi-tier load balancing architectures. Datacenter operators assign a *Virtual IP* (VIP) address to each operated service. Each VIP in a DC is associated with a set of servers providing that service. Each server has a *Direct IP* (DIP) address that uniquely identifies the server within the DC.

A LB inside the DC is a device that receives incoming connections for a certain VIP and selects a server to provide the requested service. Each connection is a Layer 4 connection (typically TCP or QUIC). For each VIP, a LB partitions the space of the connection identifiers (*e.g.*, TCP 5-tuples) across all the servers (*i.e.*, DIPs) associated with that VIP. The partitioning function is stored in the LB and is used to retrieve the correct DIP for each incoming packet.

A large-scale DC may have tens of thousands of servers and hundreds of LBs [13, 15, 32]. These LBs are often arranged into different tiers (see Fig. 1). The 1st-tier of LBs are faster and less complex than those in subsequent tiers. For example, a typical DC would use BGP routers using ECMP forwarding at the 1st-tier, followed by Layer 4 LBs, in turn followed by Layer 7 LBs and applications [20]. Similar to previous work on DC load balancing, we consider Layer 7 LBs to be at the same level as the services [13, 20, 41]. Any 1st tier LB receiving a packet directed to a VIP, performs a look up to fetch the *set* of 2nd tier LBs responsible for that VIP. It then forwards the packet towards any of these LBs. The main goal of the 1st-tier is demultiplexing the incoming traffic at the per VIP level towards their dedicated 2nd tier LBs. The 2nd-tier LBs perform two crucial operations: *(i)* guaranteeing (PCC) [32] and *(ii) load balancing* the incoming connections.

2.1 Limits of Stateless Load Balancers

Traditional stateless LBs cannot guarantee PCC. A stateless LB partitions the space of connection identifiers among the set of servers. The partitioning function is stored in the LB and does not depend on the number of active connections. Most stateless LBs, *e.g.*, ECMP [9, 19] & WCMP [53], store this partitioning in the form of an *indirection table*, which maps the output of a hash function modulo the size of the table to a specific server [3, 19, 37, 53].

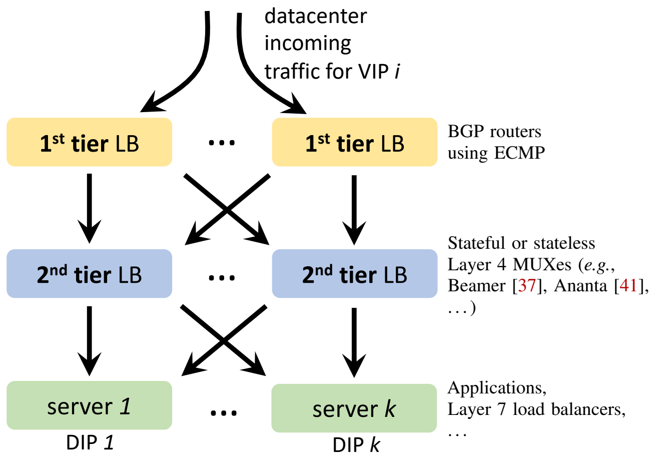

datacenter incoming traffic for VIP *i*
1st tier LB
...
1st tier LB
BGP routers using ECMP
2nd tier LB
...
2nd tier LB
Stateful or stateless Layer 4 MUXes (*e.g.*, Beamer [37], Ananta [41], ...)
server *1*
...
server *k*
DIP *1*
DIP *k*
Applications, Layer 7 load balancers, ...

Figure 1: A traditional datacenter load balancing architecture.

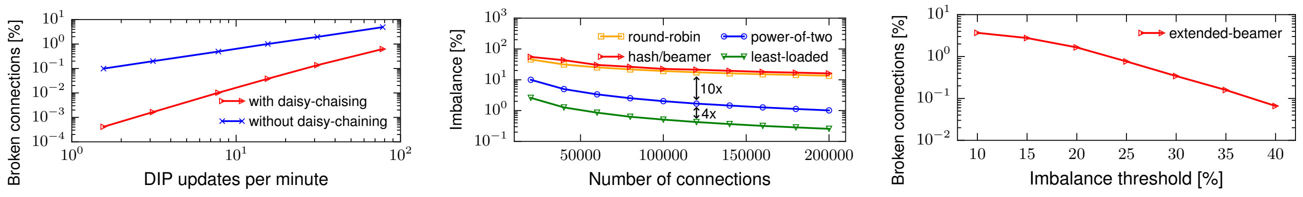
A *uniform hash* scheme maps each server to an equal number of entries in the indirection table. When a LB receives a packet, it extracts the connection identifier from the packet and feeds it as input to a hash function. The output of the hash modulo the size of the indirection table determines the index of the entry in the table where the LB can find to which server the packet should be forwarded. If the number of servers changes, the indirection table must be updated, which may cause some *existing* connections to be rerouted to the new (and wrong) server that is now associated with an entry in the table, *i.e.*, a PCC violation.

Advanced stateless LBs cannot always guarantee PCC. Beamer [37] and Faild [3] introduced *daisy-chaining* to tackle PCC. They encapsulate in the header of the packet the address of a "backup" server to which a packet should be sent when the LB hits the wrong server. This backup server is selected as the last server that was assigned to a given entry in the indirection table before the entry was remapped. PCC violations are prevented as long as *(i)* one does not perform two reconfigurations that change the same entry in the table twice (as only one backup server can be stored in the packet) and *(ii)* one can *simultaneously* reconfigure all the LBs (see [37] for an example).

Fig. 2a shows the percentage of broken connections (*i.e.*, PCC violations) with and without daisy chaining in our large-scale simulations. We used the same parameters, traffic workloads, and cluster reconfiguration events derived from previous work on real-world DC load balancing, *i.e.*, SilkRoad [32]. Namely, we simulated a cluster of 468 servers and we generated a workload using the same traffic distribution of a large web server service. We performed *DIP updates*, *i.e.* removal or additional of servers from the cluster, using different frequency distributions. SilkRoad reports that 95% of their clusters experience between 1.5 and 80 DIP updates/minute and provide distributions for the update time. We define the number of *broken connections* as the number of connections that have been mapped to *at least two* different servers during their starting and ending times. Fig. 2a shows that Beamer and Faild (plotted using the same line) still break almost 1% of the connections at the highest DIP update frequency, which may lead to an unacceptable level of service level agreement (SLA) violations [32].

Hash-based LBs cannot uniformly spread the load. We now investigate the ability of different load balancing mechanisms to uniformly spread the load across the servers for a single VIP. Similarly to the Google Maglev work [13], we define the *imbalance* of a server as the ratio between the number of connections active on that server and the average number of active connections across all servers. We also define the *system imbalance* as the maximum imbalance of any server. The imbalance of a simulation run is the *average* imbalance of the system during the entire duration of the simulation. We discuss different load metrics in Sect. 4. Using the same simulation settings as described above, we compare *(i)* Beamer [37]/Faild [3], which use a uniform hash, *(ii)* Round-Robin [50], which assigns each new connection to the next server in a list, *(iii)* Power-Of-Two [33], which picks the least loaded among two random servers, and *(iv)* Least-Loaded [50], which assigns each new connection to the server with fewest active connections. We note that Round-Robin, Power-Of-Two, and Least-Loaded require storing the connection-to-server mapping, hence they cannot be supported by Beamer/Faild. In this simulation, we do not change the size of the cluster but rather vary only the number of connections that are active at the same time in the cluster between 20K and 200K. We choose this range of active connections to induce the same imbalances (15%-30%) observed for uniform hashes in Google Maglev [13]. Fig. 2b shows the results of our simulations. We note that Beamer-like hash-based LBs outperform consistent hashing by a factor of 2x. Round-Robin outperforms a Beamer-like LB by a factor of 1.2x. When comparing these schemes with Power-Of-Two, we observe a reduction in imbalance by a factor of 10x. Finally, Least-Loaded further reduces the imbalance by an additional factor of 4*x*. These results show that a more uniform distribution of loads can be achieved by storing the mapping between connections and servers, though one still has to support PCC when the LB pool size changes. We note that today's stateful LBs [13,20,32,41] rely on different variations of uniform-hash, thus suffer from imbalances similarly to Beamer.

Beamer can reduce imbalance at the cost of a greater number of PCC violations. We tried to reduce the imbalances in Beamer by monitoring the server load imbalances and modify the entries in the indirection table accordingly. We extended Beamer with a dynamic mechanism that gets as input an imbalance threshold and remove a server from the indirection table whenever its load is above this threshold. The server is



(a) Impact of daisy-chained during DIP updates. (b) Load imbalance. (c) Dynamic Beamer imbalance.

Figure 2: Analysis of PCC-violations and load imbalances of state-of-the-art load balancers. To ease visibility, points are connected with straight lines along the x-axis.

re-added to the table when its number of active connections drops below the average. Note that, if an entry in the indirection table changes its server mapping twice, Beamer will break those existing connections that were relying on the initial state of the indirection table. Fig. 2c shows the percentage of broken connections for increasing imbalance thresholds. We set the number of active connections to 70K (corresponding to an *average* 30% imbalance in Fig. 2b). We note that guaranteeing an imbalance of at most 10% would cause 3% of all connections to break. Even with an imbalance threshold of 40% one would still observe 0.1% broken connections because of micro-bursts. Hence, even this extended Beamer cannot guarantee PCC and uniform load balancing at the same time.

2.2 Limits of Stateful Load Balancers

Stateful LBs store the connection-to-server mapping in a so-called `ConnTable` for two main reasons: *(i)* to preserve PCC when the number of servers changes and *(ii)* to enable fine-grain visibility into the flows.

Today's stateful LBs cannot guarantee PCC. Consider Fig. 1 and the case in which we add an additional stateful LB for a certain VIP. The BGP routers, which rely on ECMP, will reroute some connections to a LB than does not have the state for that connection. Thus, this LB does not know to which server the packet should be forwarded unless all LBs use an identical hash-based mechanism (and therefore experience imbalances). Therefore, existing LBs (including Facebook Katran [20], Google Maglev [13], and Microsoft Ananta [41]) rely on hashing mechanisms to mitigate PCC violations. However, this is not enough if the number of servers also changes, then some existing connections will be routed to an LB without state, hence it will hash the connection to the wrong server, thus breaking PCC.

Today's stateful LBs rely on complex and slow data structures. State-of-the-art LBs rely on cuckoo-hash tables [40] to keep per-connection mappings. These data structures guarantee constant time lookups but may require non-constant insertion time [43]. These slow insertions may severely impact the LB's throughput, *e.g.*, a throughput loss by 2x has been observed on OpenFlow switches when performing ~ 60 updates/second [34].

2.3 Service Resilience and Load Balancers

Load balancers are an indispensable component against clogging Distributed Denial of Service (DDoS) attacks, *e.g.*, bandwidth depletion at the server and memory exhaustion at the LB. Dealing with such attacks is a multi-faceted problem involving multiple entities of the network infrastructure [30], *e.g.*, firewalls, network intrusion detection, application gateways. This paper does not focus on how the LB fits into this picture but rather studies the resilience of the LB itself and the resilience its design provides to the service operation.

LBs shield servers from targeted bandwidth depletion attacks. An LB system should guarantee that the system absorb sudden bursts due to DDoS attacks with minimal impact on a service's operation. Today's LB mechanisms rely on hash-based load balancing mechanisms to provide a first pro-active level of defense, which consists in spreading connections across all servers. As long as an attacker does not reverse engineer the hash function, multiple malicious connections will be spread over the servers. A system should not allow clients to target specific servers with spurious traffic.

Stateful LBs support per-connection view at lower resilience. Stateful LBs provide fine-grained visibility into the active connections, providing resilience to the service operation, *e.g.*, by selectively rerouting DDoS flows. At the same time, stateful LBs are a trivial target of resource depletion clogging DDoS attacks: incoming spurious connections add to the connection table rapidly exhaust the limited LB memory (*e.g.*, [37]) or grow the connection table aggressively, rapidly degrading performance even with ample memory [34]. Stateless LBs can inherently withstand clogging DDoS, sustaining much higher throughput, but can only offer per-server statistics visibility to the service operation.

Having analyzed the above limitations of existing load balancers, we conclude this section by asking the following question: *"Is it possible to design a DC load balancing system that guarantees PCC, supports any realizable load balancing mechanism, and achieves similar levels of resiliency of today's state-of-the-art LBs?"*

3 The CHEETAH Load Balancer

In this section, we present CHEETAH, a load balancing system that supports arbitrary load balancing mechanisms and guarantees PCC without sacrificing performance. CHEETAH solves many of the today's load balancing problems by encoding information about the connection into a *cookie* that is added to all the packets of a connection.[1] CHEETAH sets the cookie according to any chosen and realizable load balancing mechanism and relies on that cookie to *(i)* guarantee future packets belonging to the same connection are forwarded to the same server and *(ii)* speed up the forwarding process in a stateful LB, which in turn increases the resilience of the LB. Understanding what information should be encoded into the cookie, how to encode it, and how to use this information inside a stateless or stateful LB is the goal of this section. We start our discussion by introducing the stateless CHEETAH LB, which guarantees PCC and preserves the same resilience and packet processing performance of existing stateless LBs. We then introduce the stateful CHEETAH LB, which improves the packet processing performance of today's stateful LBs, and present an LB architecture that strikes different tradeoffs in terms of performance and resilience. We stress the fact that CHEETAH does not propose a novel LB mechanism but is a building block for supporting arbitrary LB mechanisms without breaking PCC (we show the currently implemented LB mechanisms in Sect. 4).

A naïve approach. We first discuss a straightforward approach to guarantee PCC that would not work in practice because of its poor resiliency. It entails storing the identifier of a server (*i.e.*, the DIP) in the cookie of a connection. In this way, an LB can easily preserve PCC by extracting the cookie from each subsequent incoming packet. We note that such naïve approaches are reminiscent of several previous proposals on multi path transport protocols [10, 39], where the identifiers of the servers are explicitly communicated to the clients when establishing multiple subflows within a connection. There is at least one critical resiliency issue with this approach. Some clients can wait to establish many connections to the same server and then suddenly increase their load. This is highly undesired as it leads to cascade-effect imbalances and service disruptions [47].

3.1 Stateless CHEETAH LB

The stateless CHEETAH LB: encoding an opaque offset into the cookie. We now discuss how we overcome the above issues in CHEETAH. We aim to achieve the same resiliency levels[2] of today's production-ready stateless LBs (*e.g.*, Faild [3]/Beamer [37, 47]) while supporting arbitrary load balancing mechanisms and guaranteeing PCC. We

[1] We discuss legacy-compatibility issues in Sect. 4

[2] See Sect 2.3 for details of stateless/stateful load balancing resiliency.

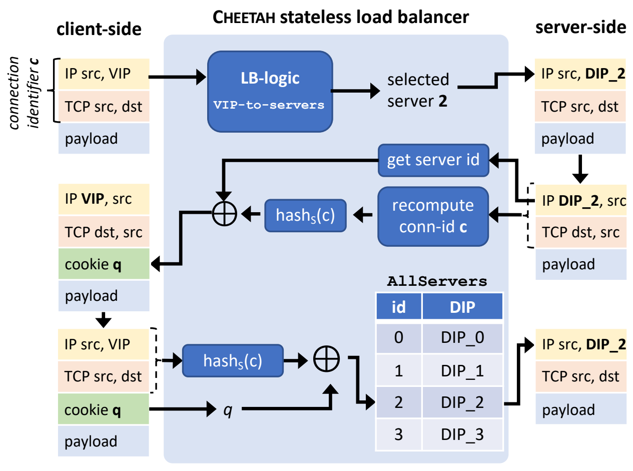


Figure 3: CHEETAH stateless LB operations.

assume a single tier LB architecture and defer the discussion of multi-tier architectures to later in this section.

The CHEETAH stateless LB keeps two different types of tables (see Fig. 3): an `AllServers` table that maps a server identifier to the DIP of the server and a `VIPToServers` table that maps each VIP to the set of servers running that VIP. The `AllServers` table is mostly static as it contains an entry for each server in the DC network. Only when servers are deployed in/removed from the DC is the `AllServers` table updated. The `VIPToServers` table is modified when the number of servers running a certain service increases/ decreases, a more common operation to deal with changes in the VIP current demands.

When the LB receives the first packet of a connection (top part of Fig. 3), it extracts the set of servers running the service (*i.e.*, with a given VIP) from the `VIPToServers` table, selects one of the servers according to any pre-configured load balancing mechanism, and forwards the packet.[3] For every packet received from a server (middle part of Fig. 3), the LB encodes an "opaque" identifier of the server mapping into the cookie for this connection. To do so, CHEETAH computes the hash of the connection identifier with a salt S (unknown to the clients), XORs it with the identifier of the server, and adds the output of the XOR to the packet header as the cookie. The salt S is the same for all connections. When the LB receives any subsequent packet belonging to this connection (bottom part of Fig. 3), it extracts the cookie from the packet header, computes the hash of the connection identifier with the salt S, XORs the output of the hash with the cookie, and uses the output of the XOR as the identifier of the server. The LB then looks up the DIP of the server in the `AllServers` table.

Stateless CHEETAH guarantees PCC. CHEETAH relies on two main design ideas to avoid breaking connections:

[3] How to implement different LB mechanisms in programmable hardware and software LBs is shown in Sect. 4.

(i) moving the state needed to preserve the mapping between a connection and its server into the packet header of the connection and *(ii)* using the more dynamic `VIPToServers` table only for the 1st packet of a connection. Subsequently, the static `AllServers` table is used to forward packets belonging to any existing connection. This trivially guarantees PCC. We defer discussion of multi-path transport protocols to Sect. 6.

Compared to existing stateless LBs Stateless CHEETAH achieves similar resiliency. Binding the cookie with the hash of the connection identifier brings one main advantage compared to the earlier naïve scheme, as an attacker must first reverse engineer the hash function of the LB in order to launch an attack targeting a specific server. This makes CHEETAH as resilient as other production-ready stateless LBs. We note that CHEETAH is orthogonal to DDoS mitigation defence mechanisms, especially when deployed in reactive mode. We further discuss CHEETAH resilience, including support for multi path transport protocols, in Sect. 6.

Stateless CHEETAH supports arbitrary load balancing mechanisms. All the reviewed state-of-the-art LBs (even stateful ones) are restricted to uniform hashing when it comes to load balancing mechanisms — as any other mechanism would break an unacceptable number of connections when the number of servers/LBs changes. In contrast, whenever a new connection arrives at a stateless CHEETAH LB, CHEETAH selects a server among those returned from a lookup in the `VIPToServers` table. The selected server may depend upon the specific load balancing mechanism configured by the service's operator. We note that the selection of the server may or may not be implementable in the data-plane. The CHEETAH LB guarantees that once the mapping connection-to-server has been established by the LB logic (not necessarily at the data-plane speed), all the subsequent packets belonging that that connection will be routed to the selected server. Since the binding of the connection to the server is stored in the packet header, CHEETAH can support LB mechanisms that go well beyond uniform hashing. For instance, an operator may decide to rely on "power of two choices" [33], which is known to reduce the load imbalance by a logarithmic factor. Another service operator may prefer a weighted round-robin load balancing mechanism that uses some periodically reported metrics (*e.g.*, CPU utilization) to spread the load uniformly among all the servers.

Lower bounds on the size of the cookie. In CHEETAH, the size of the cookie has to be at least $\log_2 k$ bits, where k is the maximum number of servers stored in the `AllServers` table. Therefore, the size of the cookie grows logarithmically in the size of the number of servers. One question is whether PCC can be guaranteed using a cookie whose size is smaller than $\log_2(k)$ and the memory size of the LB is constant. We defer proof of the following theorem to App. A.

Theorem 1. Given an arbitrarily large number of connections, any load balancer using $O(1)$ memory requires cookies of size $\Omega(\log(k))$ to guarantee PCC under any possible change in the number of active servers, where k is the overall number of servers in the DC that can be assigned to the service with a given VIP.

In App. A, we generalize the above theorem to show a certain class of advanced load balancing mechanisms, including round-robin and least-loaded, requires cookies with a size of at least $\log_2(k)$ bits even in the absence of changes in the set of active servers.

While the above results close the doors to any sublogarithmic overhead in the packet header; in practice, operators may decide to trade some PCC violations and load imbalances for a smaller sized cookie. We refer the reader to App. B for a discussion about how to implement CHEETAH with limited size cookies.

3.2 Stateful CHEETAH LB

We also designed a stateful version of CHEETAH to support a finer level of visibility into the flows than that offered by stateless LBs. A stateful LB can keep track of the behaviour of each individual connection and support complex network functions, such as rate limiters, NATs, detection of heavy-hitters, and rerouting to dedicated scrubbing devices (as in the case of Microsoft Ananta [41] and CloudFlare [30]). In contrast to existing LBs, our stateful LB guarantees PCC (inherited from the stateless design) and uses a more performant `ConnTable` that is amenable to fast data plane implementations. In the following text we say that PCC is guaranteed if a packet is routed to the correct server as long as an LB having state for its connection exists.

The stateful CHEETAH load balancer: encoding table indices in the packet header. As discussed in Sect. 2, today's stateful LBs rely on advanced hash tables, *e.g.*, cuckoo-hashing [40], to store per-connection state at the LB [32]. Such data structures offer constant-time data-plane lookups but insertion/modification of any entry in the table requires intervention of the slower control plane or complex & workload-dependant data structures (*e.g.*, Bloom filters [32], Stash-based data structures [43]), which are both complex and hard to tune for a specific workload.

We make a simple yet powerful observation about stateful tables that any insertion, modification, or deletion of an entry in a table can be greatly simplified if a packet carries information about the index of the entry in the table where its connection is stored. Since datacenters may have tens of billions of active connections, we need to devise a stateful approach where the size of the cookie is explicitly given as input. In a stateful CHEETAH LB (see Fig. 4), we store a set of m `ConnTable` tables that keep per-connection statistics and DIP mappings. We also use an equal number of `ConnStack` stacks of indices, each storing the unused entries in its corresponding `ConnTable`.

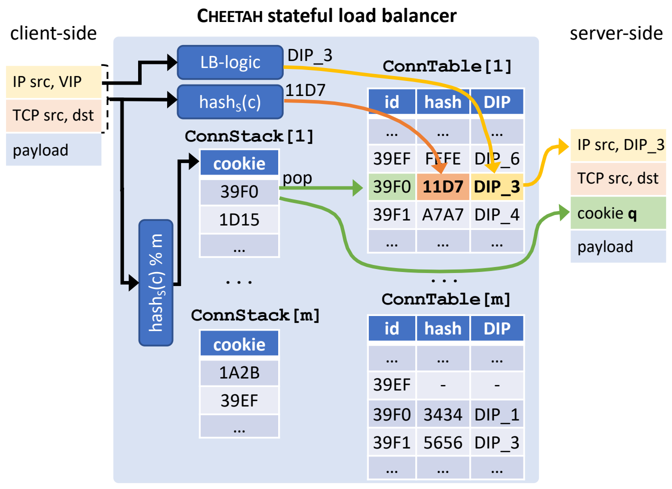


Figure 4: CHEETAH stateful LB operations for the 1st packet of a connection. We do not show the stateless cookie for identifying the stateful LB. The `VIP-to-servers` is included within the LB-logic and not shown. The server performs Direct Server Return (DSR) so the response packet does not traverse the load balancers. Subsequent packets from the client only access their index in the correspoding `ConnTable`.

For the sake of simplicity, we first assume there is only one LB and one `ConnTable` with its associated `ConnStack`, *i.e.*, $m = 1$. Whenever a new connection state needs to be installed, CHEETAH pops an index from `ConnStack` and incorporates it as part of the cookie in the packet's header. It also stores the selected server and the hash of the connection identifier with a salt S into the corresponding table entry. This hash value allows the LB to filter out malicious attempts to interfere with legitimate traffic flows, similarly to SilkRoad [32]. Whenever a packet belonging to an existing connection arrives at the LB, CHEETAH extracts the index from the cookie and uses it to quickly perform a lookup only in the `ConnTable`. Note that insertion, modification, and deletion of connections can be performed in constant time entirely in the data plane. We explain details of the implementation in Sect. 4.

The number of connections that we can store within a single `ConnTable` is equal to 2^r, where r is the size of the cookie. In practice, the size of the cookie may limit the number of connections that can be stored in the LB. We therefore present a hybrid approach that uses a hash function to partition the space of the connection identifiers into m partitions. As for any stateful table, m should be chosen high enough so the total number of entries $m * 2^r$ is suitable. The same cookie can be re-used among connections belonging to distinct partitions.

A hybrid datacenter architecture. Stateful LBs are typically not deployed at the edge of the datacenter for two reasons: they are more complex and slower compared to stateless LBs. As such, they are a weak point that could compromise the entire LB availability. Therefore, we propose a 2-tier DC architecture where the first tier consists of stateless CHEETAH LBs and the second tier consists of stateful CHEETAH LBs. The stateless LB uses the first bytes of the cookie to encode the identifier of a stateful load balancer, thus guaranteeing a connection always reaches the same LB regardless of the LB pool size. The stateful load balancer uses the last bytes of the cookie to encode per-connection information as described above.

4 Implementation

The simplicity of our design makes CHEETAH amenable to highly efficient implementations in the data-plane. We implemented stateful and stateless CHEETAH LBs on FastClick [5], a faster version of the Click Modular Router [26] that supports DPDK and multi-processing. Previous stateless systems, such as Beamer [37], have also relied on FastClick for their software-based implementation. We also implemented stateless and stateful versions of the CHEETAH LB with a weighted round-robin LB on a Tofino-based switch using P4 [6].[4] We can only make a general P4 implementation available due to Tofino-related NDAs. Both implementations are available at [4]. We first discuss the critical question of where to actually store the cookie in today's protocols and then describe the FastClick and P4-Tofino implementations.

Preserving legacy-compatibility. Our goal is to limit the amount of modifications needed to deploy CHEETAH on existing devices. Ideally, we would like to use a dedicated TCP option for storing the CHEETAH cookie into the packet header of all packets in a connection. However, this would require modifications to the clients, which would be infeasible in practice. We therefore identified three possible ways to implement cookies within existing transport protocols *without* requiring any modifications to the clients' machines: *(i)* incorporate the cookie into the `connection-id` of QUIC connections, *(ii)* encode the cookie into the least significant bits of IPv6 addresses and use IPv6 mobility support to rebind the host's address (the LB acts as a home agent), and *(iii)* embed the cookie into part of the bits of the TCP timestamp options. In this paper, we implemented a proof-of-concept CHEETAH using the TCP timestamp option as explained in App. C.[5] . We note that similar encodings of information into the TCP timestamp have been proposed in the past but require modifications to the servers [39]. The stateless CHEETAH LB can transparently translates the server timestamps with the encoded timestamps without interfering

[4]Detailed performance benchmarking of CHEETAH on the Tofino switch is subject to an NDA. The Tofino implementations follow the description of the mechanisms presented in Sect. 3, use minimal resources, and incur neither significant performance overheads nor require packet recirculation.

[5]We verified in App. C that the latest Android, iOS, Ubuntu, and MacOS operating systems support TCP timestamp options but not Windows.

with TCP timestamp related mechanisms (*i.e.*, RTT estimation and protections against wrapped sequences [8]). Therefore, no modifications are required to the servers for stateless mode unless the datacenter operator wants to guarantee Direct Server Return (DSR), *i.e.*, packets from the servers to the client do not traverse any load balancer. In that case, the server must encode the cookie into the timestamp itself. The cookie must also be sent back by the server for stateful mode, as the load balancer would not be able to find the stack index for returning traffic. Server modifications are described in C.2. We leave the implementation of CHEETAH on QUIC and IPv6 as future work. We note that a QUIC implementation would be easier and more performant since parsing TCP options is an expensive operation in both software and hardware LBs.

Table 1: TCP Options pattern

SYN packets	
MSS SAckOK Timestamp [NOP WScale]	49.86%
MSS NOP WScale NOP NOP Timestamp [SAckOK EOL]	44.49%
MSS NOP WScale SAckOK Timestamp	4.53%
Slow path	1,12%
SYN-ACK packets	
MSS SAckOK Timestamp [NOP WScale]	76.85%
MSS NOP WScale SAckOK Timestamp	18.79%
MSS NOP NOP Timestamp [SAckOK EOL]	1.69%
MSS NOP WScale NOP NOP Timestamp [SAckOK EOL]	1.55%
Slow path	1,12%
Other packets	
NOP NOP Timestamp	98.46%
NOP NOP Timestamp [NOP NOP SAck]	1.49%
Slow path	0,05%

4.1 FastClick implementation

The FastClick implementation is a fully-fledged implementation of CHEETAH that supports L2 & L3 load balancing and multiple load balancing mechanisms (*e.g.*, round-robin, power-of-2 choices, least-loaded server). The LB supports different load metrics including number of active connection and CPU utilization. The LB decodes cookies for both stateless and stateful modes using the TCP timestamp as described above, and can optionally fix the timestamp in-place if the server is not modified to do it.

Parsing TCP options. Each TS option has a 1-byte identifier, 1-byte length, and then the content value. Options may appear in any order. This makes extracting a specific option a non-trivial operation [10]. We focus on extracting the timestamp option TS_{ecr} from a packet. To accelerate this parsing operation, we performed a statistical study over 798M packets headers from traffic captured on our campus.

Table 1 shows the most common patterns observed across the entire trace for packets containing the timestamp option. The Linux Kernel already implements a similar fast parsing technique for non-SYN(/ACK) packets. We first consider non-SYN packets (*i.e.*, "Other packets" in the table). Our study shows that 99.95% of the packets have the following pattern: NOP (1B) + NOP (1B) + TimeStamp (10B) possibly followed by other fields. When a packet arrives, we can easily determine whether it matches this pattern by performing a simple 32-bit comparison and checking that the first two bytes are NOP identiers and the third one is the Timestamp id. We process the remaining 0.05% of the traffic in the slow path. We now look at SYN packets. Consider the first row in the table, *i.e.*, MSS (4B) + SAckOK (2B) + TimeStamp (10B) + SAck + EOL. To verify if a packet matches this pattern, we perform a 64-bit wildcard comparison and check that the first byte is the MSS id, the fifth byte is the SAckOK id, and the seventh byte is the TimeStamp id. We can apply similar techniques for the remaining patterns matchable with 64 bits. Some types of hosts generate packets whose patterns are wider than 64 bits, which is the limit of our x86_64 machine. We then rely on one SSE 128bit integer wildcard comparison to verify such patterns. The remaining 2.24% of patterns are handled through a standard hop-by-hop parsing following the TCP options Type-Length-Value chain. Finally, we note that we can completely avoid the more complex parsing operations for SYNs and SYN/ACKs if servers use TCP SYN cookies [12] (see App. C for more details).

Load balancing mechanisms. CHEETAH supports any realizable LB mechanisms while guaranteeing PCC. We implemented several load balancing mechanisms that will be evaluated using multiple workloads in Sect. 5.2. Among the load-aware LB mechanisms, we distinguish between metrics that can be tracked with or without coordination. Without any coordination, the LB can keep track of the number of packets/bytes sent per server and an estimate of the number of open connections based on a simple SYN/FIN counting mechanism.[6] For LB approaches that require coordination with the servers, our implementation supports load distribution based on the CPU utilization of the servers. Note that using a least-loaded server for coordination-based approaches is a bad idea as a single server will receive all the incoming connections until its load metric increases and is reported to the LB, ultimately leading to instabilities in the system. Therefore, we decided to implement the following two load-aware balancing mechanisms, which we introduced in Sect. 2: *(i)* power-of-2 choices and *(ii)* a weighted round robin (WRR). For WRR, we devised a system where the weights of the servers change according to their relative (CPU) loads. We increase the weights for servers that are underutilized depending on the difference between their load and the average server load. More formally, the number of buckets N_i assigned to server i is computed as $N_i = round(10\frac{L_{avg}}{(1-\alpha)*L_i+\alpha*L_{avg}})$ where L_i is the load of a server, and α is a factor that tunes the speed of the convergence, which we set to 0.5. A perfectly balanced system would give $N = 10$ buckets to each server. An underutilized server gets more than

[6] We envision an ad-hoc mechanism to signal closed connection between the LB and the server would make the estimate reliable in the future.

N buckets (in practice limited to $3N$) while an overloaded server gets less than N buckets (lower bounded by 2).

4.2 P4-Tofino prototype

The stateless CHEETAH LB follows exactly the description from Sect. 3.1. We store the `all-servers` and the `VIP-to-servers` tables using exact-match tables. We rely on registers, which provide per-packet transactional memories, to store a counter that implements the weighted-round-robin LB. We note that implementing other types of LB mechanisms such as least-loaded in the data-plane is non trivial in P4 since one would need to extract a minimum from an array in $O(1)$. This operation will likely requires to process the packet on the CPU of the switch. The insertion/deletion of the cookie on any subsequent non-SYN packet can be performed in the data-plane. The stateful CHEETAH LB adheres to the description in Sect. 3.2. We use P4 registers to enable the insertion of connections into the `ConnTable` at the speed of the data-plane. We store the elements of the `ConnStack` stack in an array of registers, the `ConnTable` into an array of registers, and the pointer to the head of the stack in another register.

5 Evaluation

The CHEETAH LB design allows datacenter operators to unleash the power of arbitrary load balancing mechanisms while guaranteeing PCC, *i.e.*, the ability to grow/shrink the LB and DIP pools without disrupting existing connections. In this section, we perform a set of experiments to assess the performance achievable through our stateless and stateful LBs. We focus only on evaluating the performance of the FastClick implementation.[7] All experiments scripts, including documentation for full reproducibility are available at [4].

We pose three main questions in this evaluation:

- *"How does the **cost of packet processing** in* CHEETAH *compare with existing LBs?"* (Sect. 5.1)
- *"Can we reduce **load imbalances** by implementing more advanced LB mechanisms in* CHEETAH*?"* (Sect. 5.2)
- *"How does the **PCC support** in* CHEETAH *compare with existing stateless LBs?"* (Sect. 5.3)

Experimental setting. The LB runs on a dual-socket, 18-core Intel®Xeon®Gold 6140 CPU @ 2.30GHz, though only 8 cores are used from the socket attached to the NIC. Our testbed is wired with 100G Mellanox Connect-X 5 NICs [48] connected to a 32x100G NoviFlow WB5000 switch [36]. All CPUs are fixed at their nominal frequency.

Workload generation. To generate load, we use 4 machines with a single 8-core Intel®Xeon®Gold 5217 CPU @ 3.00GHz with hyper-threading enabled using an enhanced version of WRK [17] to generate load towards the LB. We also use four machines to run up to 64 NGINX web servers (one per hyper-thread), isolated using Linux network namespaces. Each NGINX server has a dedicated virtual NIC using SRIOV, allowing packets to be switched in hardware and directly received on the correct CPU core. We generate requests from the clients using uniform and bimodal distributions, as well as the large web server service distributions already used in the simulations of Sect. 2.

[7]We argue that our Tofino implementation would perform similarly in terms of ability to uniformly distribute the load.

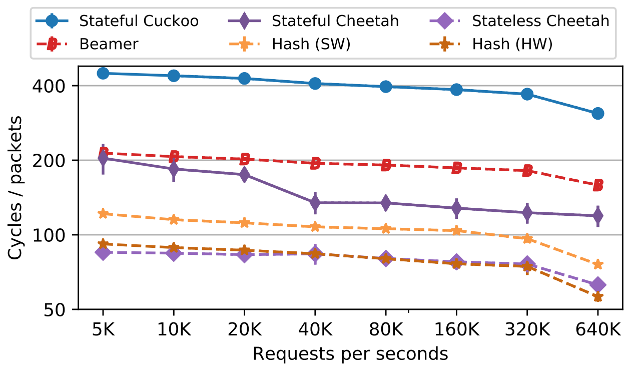


Figure 5: CPU cycles/packets for various methods. CHEETAH achieves the same load balancing performance as stateful LBs with 5x fewer cycles and only a minor penalty over hashing.

Metrics. We evaluate the imbalance among servers using both the variance of the server loads and the 99th percentile flow completion times (FCTs), where the latter one is key for latency-sensitive user applications. We measure the LB packet processing time in CPU cycles per second. Each point is the average of 10 runs of 15 seconds unless specified otherwise.

5.1 Packet Processing Analysis

We first investigate the cost in terms of packet processing time for using stateless CHEETAH. We compare it against stateful CHEETAH, a stateful LB based on per-core DPDK cuckoo-hash tables, and two hashing mechanisms, one using the hash computed in hardware by the NIC for RSS [21], and one computed in software with DPDK [29]. We also compare with a streamlined version of Beamer [38], without support for bucket synchronization, UDP, and MPTCP, thus representing a lower-bound on the Beamer packet processing cost.

Stateless CHEETAH incurs minimal packet processing costs. Fig. 5 shows the number of CPU cycles consumed by different LBs divided by the number of forwarded packets for increasing number of requests per second. We tune the request generation for a file of 8KB so that none of the clients or servers were overloaded. The main result from this experiment is that stateless CHEETAH consumes almost the same number of CPU cycles per packet as the most optimized, hardware assisted hash-based mechanism and significantly fewer cycles than stateful approaches. Beamer consumes more cycles than both CHEETAH LBs, still without bringing PCC

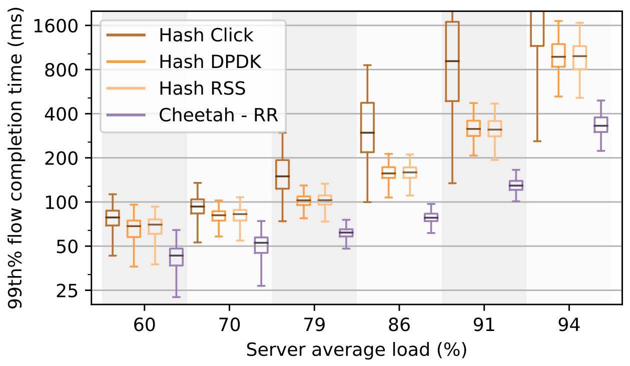


Figure 6: 99th-perc. FCT for the increasing average server load. CHEETAH achieves 2x−3x lower FCT than Hash RSS.

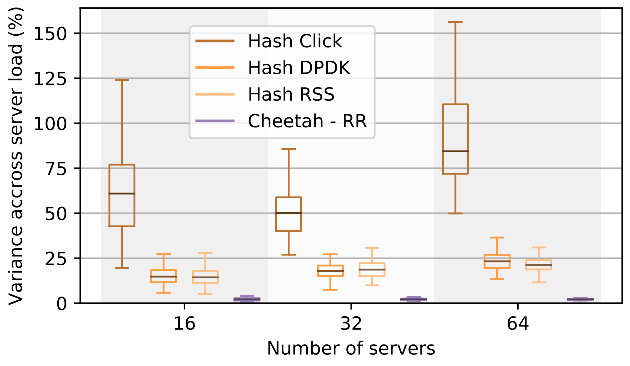


Figure 7: Variance among servers' load of various methods for an increasing number of servers. The average requests/s is 100 per server. CHEETAH, though stateless, allows a near-perfect load spreading.

guarantee (see Sect. 5.3). This is mainly due to the operation of encapsulating the backup server into the packet header and the more compute-intensive operations needed by Beamer to lookup into a bigger "stable hashing" table. Finally, we note that, with the web service requests size distribution, each methods only need 4 CPU cores to saturate the 100Gbps link.

Stateful CHEETAH outperforms cuckoo-hash based LBs. We also note in Fig. 5 the improvements in packet processing time of stateful CHEETAH (which uses a stack-based `ConnTable` table) compared to the more expensive stateful LBs using a cuckoo-hash table. Stateful CHEETAH achieves performance close to a stateless LB and a factor of $2-3$x better that cuckoo-hash based LBs.

Dissecting stateless CHEETAH performance. The key insight into the extreme performance of CHEETAH is that the operation of obfuscating the cookie only adds less than a 4-cycle hit. We in fact rely on the network interface card hardware to produce a symmetric hash (*i.e.*, using RSS). We expect the advent of SmartNICs as well as QUIC and IPv6 implementations, which have easier-to-parse headers, to perform even better. We note that our stateless CHEETAH implementation uses server-side TCP timestamp correction (see Sect. 4), which only imposes a 0.2% performance hit over the server processing time. If we were to use LB-side timestamp correction, we observe that the stateless CHEETAH modifies the timestamp MSB on the LB in just 30 cycles per packet performance hit. To summarize, stateless CHEETAH brings the same benefits as stateful LBs (in terms of load balancing capabilities) in addition to PCC guarantees at basically the same cost (and resilience) of stateless LBs.

5.2 Load Imbalance Analysis

We now assess the benefits of running CHEETAH using a non-hash-based load balancing mechanism and compare it to different uniform hash functions (similarly to those implemented in Microsoft Ananta [41], Google Maglex [13], Beamer [37], and Faild [3]). We stress that we do not propose novel load balancing mechanisms but rather showcase the potential benefits of a load balancer design that supports any realizable load balancing mechanisms. We only evaluate stateless CHEETAH as the load imbalance does not depend on the stored state (and would result in similar performance).

In this experiment, each server performs a constant amount of CPU-intensive work to dispatch a 8KB file. The generator makes between 100 and 200 requests per server per second on average depending on our targeted system load. Given this workload and service type, we expect an operator to choose a uniform round-robin LB mechanism to distribute the load.

CHEETAH significantly improves flow completion time. Fig. 6 compares CHEETAH with round-robin and hash-based LB mechanisms with 64 servers. We consider three hash functions: Click [26], DPDK [29], and the hardware hash from RSS. We stress the fact that these hash-based functions represent the quality of load balancing achievable by existing stateless (*e.g.*, Beamer [37]) and stateful LBs (*e.g.*, Ananta [41]) LBs. We measure the 99th percentile flow completion time (FCT) tail latency for the increasing average server load. We note that CHEETAH reduces the 99th percentile FCT by a factor of $2-3$x compared to the best performing hash-based mechanism, *i.e.*, Hash RSS.

CHEETAH spreads the load uniformly. To understand why CHEETAH achieves better FCTs, we measure the variance of the servers' load over the experiment for an average server load of 60% and 16, 32, and 64 servers. Fig. 7 shows that (as expected) the variance of RR is considerably smaller than hash-based methods. This is because the load balancer iteratively spreads the incoming requests over the servers instead randomly spreading them. In this specific scenario, CHEETAH allows operators to leverage RR, which would otherwise be impossible with today's load balancers. Fig. 7 also shows that the quality of the hash function is important

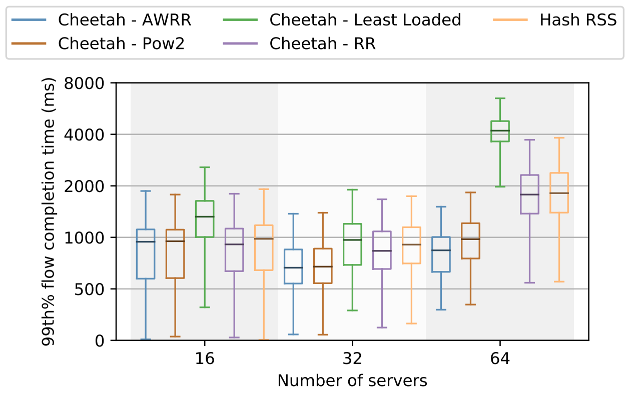


Figure 8: Evaluation of multiple load balancing methods for a bimodal workload. Both AWRR and Pow2 outperform Hash RSS by a factor of 2.2 and 1.9 respectively with 64 servers.

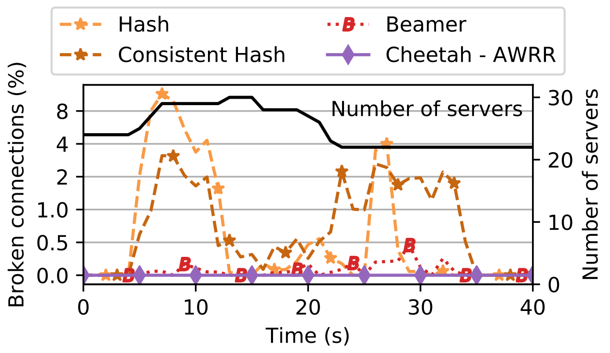


Figure 9: Percentage of broken requests while scaling the number of servers. Cheetah guarantees PCC whereas hashing breaks up to 11% of the connections, consistent hashing 3% and Beamer up to 0.5%.

as the default function provided in Click does not perform well. In contrast, the CRC hash function used by DPDK is comparable to the Toeplitz based function used in RSS [28]. Moreover, the RSS function has the advantage of being performed in hardware.

CHEETAH improves FCT even with non-uniform workloads. Fig. 8 shows the tail FCT for a bimodal workload, where 10% of requests take 500ms to be ready for dispatching and the remaining ones take a few hundred microseconds. In this scenario, some servers will be loaded in an unpredictable way thus creating a skew that requires direct feedback from the servers to solve. We can immediately see that RR with 64 servers leads to very high FCTs. We evaluate three ways to distribute the incoming requests according to the current load (see Sect. 4.1): automatic weighted round robin (AWRR), power of two choices (Pow2), and the least loaded server. Each server piggybacks its load using a monitoring Python agent on the server that reports its load through an HTTP channel to the LB at a frequency of 100Hz, though experimental results showed similar performance at 10Hz. Least loaded performs poorly since it sends all the incoming requests to the same server for 10ms, overloading a single server. Pow2 and AWRR spread the load more uniformly as the LB penalizes those servers that are more overloaded. Consequently, both methods reduce the FCT by a factor of two compared to Hash RSS with 64 servers. These experiments demonstrate the potential of deploying advanced load balancing mechanisms to spread the service load.

5.3 PCC Violations Analysis

We close our evaluation by demonstrating the key feature of the stateless CHEETAH LB, *i.e.*, its ability to avoid breaking connections while changing the server and/or LB pool sizes. We compare CHEETAH against Hash RSS, consistent hashing, and Beamer. We start our experiment with a cluster consisting of 24 servers. We tune a python generator [4] to create 1500 requests/s, increasing following a sinusoidal load to 2500 requests/s and descending back to 1500 over the 40 seconds of the experiment. The size and duration of the requests are served using the web server distribution. We iteratively add 7 servers to the pool as the load increases. We then drain 8 servers when the rate goes down. Fig. 9 shows the percentage of broken requests over completed requests every second over time. Some connections gets accounted as broken dozen of seconds later as clients continue sending retransmission before raising an error. Compared to Beamer, Cheetah not only achieves better load balancing with AWRR (Sect. 5.2), but it also does not break any connection.

6 Frequently Asked Questions

Does CHEETAH preserve service resilience compared to existing LBs? Yes. We first discuss whether a client can clog a server. A client generating huge amounts of traffic using the same connection identifier can be detected and filtered out using heavy-hitter detectors [41]. This holds for any stateless LBs, *e.g.*, Beamer [37]. A more clever attack entails reverse engineering the salted hash function and deriving a large number of connection identifiers that the LB routes to the same (specific) server, possibly with spoofed IP addresses. To do so, an attacker needs to build the $(conn.id, cookie) \mapsto server$ mapping. This requires performing complex measurements to verify whether two connection IDs map to the same server. Given that CHEETAH uses the same hash function of any existing LB (which is not cryptographic due to their complexity [3]), reverse engineering this mapping will be as hard as reverse engineering the hash of the existing LBs. As for the resilience to resource depletion, we note from Fig. 5 that stateless (stateful) CHEETAH has similar (better) packet processing times of today's stateless (stateful) LBs. Thus, we argue that CHEETAH achieves the same levels of resilience of today's existing LB systems.

Does CHEETAH make it easy to infer the number of servers? Not necessarily. A 16-bit cookie permits at least an order of magnitude more servers than the number of servers used to operate the largest services [32]. If this is still a concern, one can hide the number of servers by reducing the size of the cookie and partitioning the connection identifier space similarly to our stateful design of CHEETAH.

Does CHEETAH support multipath transport protocols? Yes. In multipath protocols, different sub-connection identifiers must be routed to the same DIP. Previous approaches exposed the server's id to the client [10, 39]; however, this decreases the resilience of the system decreases. CHEETAH can use a different permutation of `AllServers` for each additional *i*'th sub-connection. Clients inform the server of the new sub-connection identifier to be added to an existing connection. The server replies with the cookie to be used using the *i*'th `AllServers` table. This keeps the resilience of the system unchanged compared to the single path case.

7 Related Work

There exists a rich body of literature on datacenter LBs [2, 7, 11, 13, 16, 18–20, 22–25, 32, 37, 41, 52, 53]. We do not discuss network-level DC load balancers [2, 7, 16, 18, 22, 24, 25, 52], whose goal is to load balance the traffic within the DC network and do not deal with per-connection-consistency problems.

Stateless LBs. Existing stateless LBs rely on hash functions and/or "daisy chaining" techniques to mitigate PCC violations (2), *e.g.*, ECMP [19], WCMP [53], consistent hashing [23], Beamer [37], and Faild [3]. The main limitation of such schemes is the suboptimal balancing of the server loads achieved by the hash function, which is known to grow exponentially in the number of servers [49]. Shell [42] proposed a similar use of the timestamp option as a reference to an history of indirection tables, which comes at both the expense of memory and low-frequency load rebalancing.

QUIC-LB [11] is a high-level design proposal at the IETF for a stateless LB that leverages the `connection-id` of the QUIC protocol for routing purposes. While sharing some similarities to our approach, QUIC-LB *(i)* does not present a design of a stateful LB that would solve cuckoo-hash insertion time issues, *(ii)* does not evaluate the performance obtainable on the latest generation of general-purpose machines, *(iii)* relies on the modulo operation with an odd number to hide the server from the client, an operation that is not supported in P4, and *(iv)* does not discuss multi path protocols. We note that our cookie can be implemented as the 160-bit `connection-id` in QUIC, which is also easier to parse than the TCP timestamp option. Encoding the connection-to-server mapping has recently been briefly discussed in an editorial note without discussing LB resilience, stateful LBs, or implementing and evaluating such a solution [31].

Several stateless load balancers that support multi path transport protocols have been proposed in the past. Such load balancers guarantee all the subflows of a connection are routed to the same server by explicitly communicating an identifier of the server to the client [10, 39]. These approaches may be exploited by malicious users to cause targeted imbalances in the system, which is prevented in CHEETAH thanks to using distinct hashes for the subflows (see Sect. 6).

Stateful LBs. Existing stateful LBs store the connection-to-server mapping in a cuckoo-hash table [13, 15, 20, 32, 41] (see Sect. 2). These LBs still rely on hash-based LB mechanisms — as these lead to fewer PCC violations when changing the number of LBs. In contrast, CHEETAH decouples PCC support from the LB logic, thus allowing operators to choose any realizable LB mechanism. Moreover, cuckoo-hash tables suffer from slow (non-constant) insertion time. FlowBlaze [43] and SilkRoad [32] tackled this problem using a stash-based and bloom-filter-based implementations, respectively. Yet, both solutions cannot guarantee insertions in constant-time: FlowBlaze relies on a stash that may be easily filled by an adversary while SilkRoad is limited by both the size of the Bloom Filter and the complexity of the implementation. CHEETAH uses a constant-time stack that is amenable to fast implementation in the dataplane. Existing stateful LBs also suffer from the fact that the 1st-tier of stateless ECMP LBs reshuffle connections to the wrong stateful LB when the number of LBs changes. In contrast, 1st-tier stateless CHEETAH guarantees connections reach the correct stateful LB regardless of changes in the LB pool size.

8 Conclusions

We introduced CHEETAH, a novel building block for load balancers that guarantees PCC and supports any realizable LB mechanisms. We implemented CHEETAH on both software switches and programmable ASIC Tofino switches. We consider this paper as a first step towards unleashing the power of load balancing mechanisms in a resilient manner. We leave the question of whether one can design novel load balancing mechanisms tailored for Layer 4 LBs as well as deployability with existing middleboxes as future work.

Acknowledgements

We would like to thank our shepherd Jitendra Padhye and the anonymous reviewers for their invaluable feedback. The 7th author thanks the participants at the Dagstuhl workshop on "Programmable Network Data Planes". This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 770889). This work was also funded by the Swedish Foundation for Strategic Research (SSF). The work was also supported in parts by a KAW Academy Fellowship and an SSF Framework Grant.

References

[1] Alexa. The top 500 sites on the web. URL: https://www.alexa.com/topsites.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. *SIGCOMM Comput. Commun. Rev.*, 44(4), August 2014.

[3] Joao Taveira Araujo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. Balancing on the edge: Transport affinity without network state. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[4] Cheetah authors. Github - cheetah source code, 2020. https://github.com/cheetahlb/. URL: https://github.com/cheetahlb/.

[5] Tom Barbette. Github - FastClick, 2015. https://github.com/tbarbette/fastclick. URL: https://github.com/tbarbette/fastclick.

[6] Barefoot. Tofino: World's Fastest P4 Programmable Ethernet Switch ASIC, September 2019. URL: https://www.barefootnetworks.com/products/brief-tofino/.

[7] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, 2011.

[8] David Borman, Robert T. Braden, Van Jacobson, and Richard Scheffenegger. TCP Extensions for High Performance. RFC 7323, September 2014. URL: https://rfc-editor.org/rfc/rfc7323.txt, doi:10.17487/RFC7323.

[9] Marco Chiesa, Guy Kindler, and Michael Schapira. Traffic engineering with equal-cost-multipath: An algorithmic perspective. *IEEE/ACM Trans. Netw.*, 25(2):779–792, 2017. URL: https://doi.org/10.1109/TNET.2016.2614247, doi:10.1109/TNET.2016.2614247.

[10] Fabien Duchene and Olivier Bonaventure. Making multipath TCP friendlier to load balancers and anycast. In *25th IEEE International Conference on Network Protocols, ICNP 2017, Toronto, ON, Canada, October 10-13, 2017*, pages 1–10, 2017.

[11] Martin Duke. QUIC-LB: Generating Routable QUIC Connection IDs. Internet-Draft draft-duke-quic-load-balancers-04, Internet Engineering Task Force, May 2019. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-duke-quic-load-balancers-04.

[12] Wesley Eddy. TCP SYN Flooding Attacks and Common Mitigations. RFC 4987, August 2007. URL: https://rfc-editor.org/rfc/rfc4987.txt, doi:10.17487/RFC4987.

[13] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, 2016.

[14] Pierre Fraigniaud and Cyril Gavoille. Memory requirement for universal routing schemes. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 223–230, 1995.

[15] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[16] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, 2017.

[17] Will Glozer. WRK. URL: https://github.com/wg/wrk.

[18] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 45(4), August 2015.

[19] Christian Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, November 2000. URL: https://rfc-editor.org/rfc/rfc2992.txt, doi:10.17487/RFC2992.

[20] Christian Hopps. Katran: A high performance layer 4 load balancer, September 2019. URL: https://github.com/facebookincubator/katran.

[21] Intel. Receive-Side Scaling (RSS), 2016. URL: http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf.

[22] Nanxi Kang, Monia Ghobadi, John Reumann, Alexander Shraer, and Jennifer Rexford. Efficient traffic splitting on commodity switches. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, 2015.

[23] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, 1997.

[24] Naga Katta, Mukesh Hira, Aditi Ghag, Changhoon Kim, Isaac Keslassy, and Jennifer Rexford. Clove: How i learned to stop worrying about the core and love the edge. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, 2016.

[25] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, SOSR '16, 2016.

[26] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000. URL: http://doi.acm.org/10.1145/354871.354874, doi:10.1145/354871.354874.

[27] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *International Journal of Computer Mathematics*, 2(1-4):157–168, 1968.

[28] Hugo Krawczyk. New hash functions for message authentication. In *Proceedings of the 14th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'95, 1995.

[29] Linux Foundation. Data plane development kit (DPDK), 2015. http://www.dpdk.org. URL: http://www.dpdk.org.

[30] Marek Majkowski. SYN packet handling in the wild, January 2018. URL: https://blog.cloudflare.com/syn-packet-handling-in-the-wild/.

[31] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on load distribution and the role of programmable switches. *SIGCOMM Comput. Commun. Rev.*, 49(1):18–23, February 2019. URL: https://doi.org/10.1145/3314212.3314216, doi:10.1145/3314212.3314216.

[32] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, 2017.

[33] Michael David Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California, Berkeley, 1996. AAI9723118.

[34] Felician Nemeth, Marco Chiesa, and Gabor Retvari. Normal forms for match-action programs. In *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '19, 2019.

[35] NetApplications. Market share for mobile, browsers, operating systems and search engines | NetMarketShare. URL: https://bit.ly/37iRNOK.

[36] NoviFlow. Katran: A high performance layer 4 load balancer, September 2019. URL: https://noviflow.com/noviswitch/.

[37] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[38] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with Beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, Renton, WA, April 2018. USENIX Association. URL: https://www.usenix.org/conference/nsdi18/presentation/olteanu.

[39] Vladimir Andrei Olteanu and Costin Raiciu. Datacenter scale load balancing for multipath transport. In *Proceedings of the ACM SIGCOMM Workshop on Hot topics in Middleboxes and Network Function Virtualization, HotMiddlebox@SIGCOMM 2016, Florianopolis, Brazil, August, 2016*, pages 20–25, 2016.

[40] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2), May 2004.

[41] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, 2013.

[42] Benoît Pit-Claudel, Yoann Desmouceaux, Pierre Pfister, Mark Townsley, and Thomas Clausen. Stateless load-aware load balancing in p4. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 418–423. IEEE, 2018.

[43] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[44] Statcounter. Desktop Operating System Market Share Worldwide. URL: https://gs.statcounter.com/os-market-share/desktop/worldwide.

[45] Statcounter. Desktop vs Mobile vs Tablet Market Share Worldwide. URL: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet.

[46] Statcounter. Mobile Operating System Market Share Worldwide. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide.

[47] Helen Tabunshchyk. Super Fast Packet Filtering with eBPF and XDP, December 2017. URL: https://bit.ly/2mpoIyO.

[48] Mellanox Technologies. ConnectX®-5 EN Single/Dual-Port Adapter Supporting 100Gb/s Ethernet. URL: https://www.mellanox.com/page/products_dyn?product_family=260&mtag=connectx_5_en_card.

[49] Vladimir P. Chistyakov Valentin F. Kolchin, Boris A. Sevastyanov. *Random allocations*. Washington: Winston, 1978.

[50] John Carl Villanueva. Comparing Load Balancing Algorithms, June 2015. URL: https://www.jscape.com/blog/load-balancing-algorithms.

[51] Weikun Wang and Giuliano Casale. Evaluating weighted round robin load balancing for cloud web services. In *16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2014, Timisoara, Romania, September 22-25, 2014*, pages 393–400, 2014.

[52] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient datacenter load balancing in the wild. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, 2017.

[53] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

APPENDIX

A Proof of Theorem 1

Theorem 1. Given an arbitrarily large number of connections, any load balancer using $O(1)$ memory requires cookies of size $\Omega(\log(k))$ to guarantee PCC under any possible change in the number of active servers, where k is the overall number of servers in the DC that can be assigned to the service with a given VIP.

Proof sketch. We prove the statement of the theorem in the widely adopted Kolmogorov descriptive complexity model [27]. We leverage similar techniques used in the past to demonstrate a variety of memory-related lower bounds for shortest-path routing problems [14].

Let R be the set of all the possible connection identifiers. Let C be the set of all possible cookies. Let $S = \{s_1, \dots, s_k\}$ be the set of servers. We assume $|R| \gg |S|$, which is the most interesting case in real-world datacenters. Suppose, by contradiction, that there exists an LB which uses $O(1)$ memory with cookies of size smaller than $\log(k)$ bits that guarantees under any arbtirary number of changes in the subset of active servers $A \subseteq S$. For any possible set of active servers A, the LB maps a new incoming connection identifier $r \in R$ to a certain server $s \in S$, *i.e.*, the LB logic maps incoming connections using an arbitrarily function $f : R \times 2^A \to S$.

Let us now restrict our focus to the $|S|$ distinct sets of active servers in which only a single server is active, *i.e.*, $A_1 = \{s_1\}, \dots, A_k = \{s_k\}$. Depending on the time instance when a connection $r \in R$ arrives, the connection may be mapped to any of these servers. The mapping must be preserved for the entire duration of the connection, which means the LB must be able to forward any future packet belonging to r regardless of the current set of active servers. Therefore, the LB must be able to distinguish among $|R| \times |S|$ distinct possible mappings between connections and servers. Consider our cookie with l bits, where $l < |S| = \log(k)$. This information allows us to distinguish among $|R| \times l$ possible mappings, which leaves $|R| \times (|S| - l) = O(|R||S|)$ mappings to the LB memory. This is a contradiction since we assumed the LB uses a memory of $O(1)$. □

It is trivial to verify that the above theorem holds even if one wants to implement an advanced LB mechanism, *e.g.* round-robin, least-loaded, even without allowing any changes in the number of servers.

B Constant-size cookies

Minimizing PCC violations with constant-size cookies: keeping a history at the LB. A simple way to deal with changes in the number of active servers when using a uniform-hash load balancing mechanism is to encode in the cookie a *tag* that can be used by the LB to uniquely identify a previous configuration of active servers. The LB stores the last n configurations of active servers and identifies them using $\log_2(n)$ bits, which are encoded into the cookie. Thus, as long as a connection is mapped to a server that existed in any of the last n configurations, the connection will be unbroken. The LB can modify the cookie associated with a connection to use "fresher" cookies. [8]. Clearly, if an operator drains a server, *i.e.*, purposely does not assign new connections to it, any remaining connections will be broken after n changes in the tag.

Minimizing load imbalances using constant-size cookies: adding a hash function index to the cookie. A common technique to reduce load imbalances in a system is to rely on "power-of-two" choices mechanisms to map a request to a server. When the LB receives a new incoming connection, it computes the hash of the connection identifier using two different hash functions h_1 and h_2, whose outputs are used to select two servers. The LB then compares the load of the two selected servers and maps the incoming connection to the least loaded server among these two (according to some definition of "load"). The main issue in the DC context with power-of-two choice load balancing mechanisms is that the LB needs to remember for each connection, which of the two servers was used to serve the connection, that is, the LB must be stateful. CHEETAH can support a power-of-two choices LB mechanism simply by storing in the cookie a single bit that identifies which of the two hash functions is to be used for that connection. Similarly, multiple hashes could be supported by increasing the size of the cookie logarithmically in the number of hash functions. We note that when the number of servers change, connections may break. We refer the reader to the previous paragraph on how to deal with PCC violations with constant-size cookies.

C TCP timestamp encoding

We decided to encode the CHEETAH cookie into the 16 most significant bits of the TCP timestamp. We acknowledge that alternative ways to encode information into the TCP timestamps are possible [39] but require modifications to the servers.

Encoding cookies in the TCP timestamp option. TCP end-hosts use TCP timestamp options to estimate the RTT of the connection. The timestamp consists of a 64-bit pair (TS_{val}, TS_{ecr}), where TS_{val} and TS_{ecr} are the 32-bit timestamps set by the sender and receiver of the packet, respectively. When an end-host receives a packet, it echoes the TS_{val} back to the other end-host by copying it into the TS_{ecr}. We leverage the TS_{val} to carry the CHEETAH cookie on every packet directed towards the client, which will echo it back as the TS_{ecr}. We encode the cookie in the 16 most significant bits of the 32-bit TS_{val} for every packet directed towards a client.[9] We must therefore fix the original 16 most significant bit of the TS_{ecr} before it is processed by the TCP stack of the server. This can be done in the load-balancer or on the server itself. Our measurements of the top 100 ranked Alexa websites [1] reported in App. D shows that the minimal unit of a timestamp is 1 millisecond. This means that the least significant 16 bits of the timestamp would wrap up every 2^{16}ms.

TCP timestamps are mostly supported in today's OSes. We ran a small experiment to verify whether today's client devices support the echoing of TCP timestamp options back to the servers. We tested the latest OSes available in both recent smartphones and desktop PCs: Google Android 9, iOS 13, Ubuntu 18.04, Microsoft Windows 10, and MacOS 10.14. We observed that all except Microsoft Windows correctly negotiate and echo the TCP timestamp option when the server requires to use it. Based on some recent measurements, more than 98% of the smartphone and tablet devices are either using Android or iOS [46]. Smartphone devices are the most common type of devices, representing 53% of all devices [45]. For desktop devices, Windows is the predominant OS with over 75% of the desktop share whereas MacOS represent a 16% of this share [44]. For Windows desktop devices, a cloud operator can either encode the cookie in the QUIC header (69% of the Windows users use Google Chrome, compatible with QUIC [35]), IPv6 address, or install stateful information into the LB for these devices.

C.1 Fixing the timestamp in the load balancer

For each server, we keep in memory two versions of the 16 most significant bits (MSB) replaced by our cookie: the current one and the previous one. We use one bit of the cookie to remember the version of the original MSB for every given packet sent to the client. When a wrap up of the timestamp happens, we set the oldest MSB bucket to the new MSB timestamp of the server, and we change the version bit of outgoing packets to designate that one. When a packet is received, the cookie is read, then the original MSB given by the version bit found in the packet reader is rewritten back in

[8]This operation requires a bit more than one Round-Trip-Time (RTT) to update the client from the LB (the packet must first be processed by the server, which has to send an acknowledgment to the client with the updated cookie

[9]Timestamp options have already been used in the past for protecting against SYN flood attacks, *e.g.*, TCP SYN cookies [12]. We note that several large cloud networks completely disable TCP SYN cookies and rely on different mechanisms to handle SYN flood attacks [30], thus CHEETAH would not cause deployment issues. We note that an alternative implementation of CHEETAH, in which the mapping with the server is only performed at the end of the 3-way handshake does not prevent the cloud provider from using TCP SYN cookies. This solution requires all the servers to use exactly the same parameters when generating the TCP SYN cookies and is left as a minor future work extension.

the timestamp of the packet. To avoid having clients sending very old cookies, we rely on TCP keepalives set to 25 seconds, which allows us to both detect timestamp wraps at the LB[10] and deal with TCP PAWS packet filters[11].

C.2 Fixing the timestamp on the server

We modified the Linux Kernel 5.1.5 (available at [12]) to enable the same mechanism directly on the server. Doing so increase the server packet processing time by only 0.2%, as the timestamp is parsed in any cases on the server. This enables the server to keep the value of the cookie, and directly encode the cookie in the TS_{val}. Such returning packet do not need to go through the load balancer and allows the use of DSR. Having the cookie on both sides of the load-balancer is also needed for the stateful implementation.

D Alexa Top100 Timestamp Measurements

We ran a comprehensive set of measurements to determine the granularity of the TCP timestamp unit utilized by the largest service providers according to the Alexa Top100 ranking [1]. We downloaded large files from each the top 15 ranked web sites and extracted both the TCP timestamp TS_{val} options and the client side timestamp. We then computed the difference between the TCP last and first timestamps and divided this amount by the different between the client measured last and first (non-TCP) timestamps. The result is the granularity of the server-side TCP timestamp unit. We report the results in Table 2. All the service providers using TCP timestamps have a granularity of at least 1ms. This means the timestamp wraps every $2^{16} \approx 65$ seconds when using CHEETAH to support these services.

Table 2: Measured TCP timestamp granularity for different websited. Some service providers do not use TCP timestamp options.

Web site	**TS granularity**	**Method**
drive.google.com	1ms	gdown
dropbox.com	1ms	wget
twitch.tv	1ms	watch video
weobo.com	1ms	watch video
bilibili.com	N.A.	-
pan.baidu.com	N.A.	-
reddit.com	4ms	watch video
qq.com	4ms	watch video
instagram.com	4ms	watch video
onedrive.live.com	N.A.	-
facebook.com	1ms	watch video
twitter.com	N.A.	-
imdb.com	10ms	watch video

[10]Detecting a wrap at the server is a straightforward operation. To detect a wrap at the LB, we need to guarantee the server sends a packet at least once every 2^{15}ms$\approx$ 33s.This would typically be the case for every real-world Internet service.

[11]We note that flipping the MSB of the timestamp (*i.e.* the version bit) every time a wrap is detected may create problems with PAWS [8]. PAWS is a TCP mechanism that discards packets with TCP timestamps that are not "fresh", *i.e.*, a timestamp is considered *old* if the difference between the latest received timestamp and the newest received timestamp is smaller than 2_{31}. To avoid enabling this condition, it is sufficient to guarantee the server keeps a TCP keep-alive timer of $(2^{15} - \max RTT)$ milliseconds. Assuming a $\max RTT$ of 5 seconds, we set the keep-alive to a conservative value of 25 seconds. With 100K connections, the bandwidth overhead is just 0.00002% on a 100Gbps server interface.

[12]https://github.com/cheetahlb/linux

Programmable Calendar Queues for High-speed Packet Scheduling

Naveen Kr. Sharma[*] *Chenxingyu Zhao*[*] *Ming Liu*[*] *Pravein G Kannan*[†] *Changhoon Kim*[‡]
Arvind Krishnamurthy[*] *Anirudh Sivaraman*[§]

Abstract

Packet schedulers traditionally focus on the prioritized transmission of packets. Scheduling is often realized through coarse-grained queue-level priorities, as in today's switches, or through fine-grained packet-level priorities, as in recent proposals such as PIFO. Unfortunately, fixed packet priorities determined when a packet is received by the traffic manager are not sufficient to support a broad class of scheduling algorithms that require the priorities of packets to change as a function of the time it has spent inside the network. In this paper, we revisit the Calendar Queue abstraction and show that it is an appropriate fit for scheduling algorithms that not only require prioritization but also perform dynamic escalation of packet priorities. We show that the calendar queue abstraction can be realized using either dataplane primitives or control-plane commands that dynamically modify the scheduling status of queues. Further, when paired with programmable switch pipelines, we can realize programmable calendar queues that can emulate a diverse set of scheduling policies. We demonstrate the power of this abstraction using three case studies that implement variants of LSTF, Fair Queueing, and pFabric in order to provide stronger delay guarantees, burst-friendly fairness, and starvation-free prioritization of short flows, respectively. We evaluate the benefits associated with these scheduling policies using both a custom simulator and a small-scale testbed.

1 Introduction

Many network scheduling algorithms today require a notion of *dynamic priority*, where the priority of individual packets within a flow varies over the lifetime of the flow. These packet priorities generally change with either the amount of bytes sent by the flow, how fast the flow is transmitting, or time spent by the packet inside the network. Such scheduling algorithms enable richer application-level prioritization and performance guarantees, such as shortest-job first to minimize average flow completion time, earliest deadline first to enable timely delivery of all messages, or fair-queueing, as illustrated by a long list of proposed scheduling algorithms (e.g., pFabric [4], PIAS [5], WFQ [14], FIFO+ [13], LSTF [17], and EDF [18]).

Switch-level support for multiple fine-grained priority levels (as in PIFO [28], pHeap [7]) can aid the realization of these scheduling algorithms. However, there are still several challenges in faithfully implementing these algorithms efficiently and at line rate. First, implementing strict and fine-grained priority levels is expensive, especially at scale involving high bandwidths and hundreds or thousands of unique flows at multiple terabits per second. Second, and more crucially, existing switch support for priorities does not allow for dynamic changes to the priority of a packet during its stint inside the switch buffer. Fixed packet priorities cannot effectively emulate the *ageing* property required by many of the scheduling algorithms, wherein the priority of a packet increases with the time it spends inside a queue.

Consider, for instance, the Least Slack Time First (LSTF [17]) scheduling discipline wherein each packet maintains a delivery deadline, and the switch emits from its buffer the packet with the least slack at a given instant. LSTF cannot be realized using fixed packet priorities that are determined when the packets are inserted into a priority queue. Notably, given a packet that has a deadline of $current\_time + slack$, a switch scheduler that maps this deadline to a priority level would quickly exhaust a *finite* number of priority levels. As long as there are packets buffered inside the switch, packets received with later deadlines would have to be progressively assigned lower priority levels, and the switch will eventually run out of priority levels to use for incoming packets.

In this paper, we argue that what is needed is a scheduling mechanism that supports both prioritization and implicit escalation of a packet's priority as it spends more time inside the switch buffer. We observe that a mechanism similar to Calendar Queues [10] would be a more appropriate fit for implementing these scheduling algorithms. Calendar Queues allow events (or packets) to be enqueued at a priority level or rank corresponding to a future time, and this rank gradually changes as time moves forward. A scheduling algorithm simply decides how far in the future a packet must be processed and then controls how time is advanced, say based on the switch's physical clock (i.e., *physical time*) or the number of communication rounds across flows (i.e., *logical time*).

Calendar queues have certain properties that make them amenable to efficient hardware realization, especially on upcoming programmable switch hardware. A calendar queue consists of multiple physical switch queues, and at any point in time, only one of the queues in a calendar queue is active. Further, a calendar queue imposes a fixed rotation order for activating queues. We describe how the activation of queues in a fixed order can be achieved by periodically modifying the priority and active status of queues, either through dataplane primitives expected in future programmable switches or through today's control-plane operations, albeit at a higher

[*]University of Washington
[†]School of Computing, National University of Singapore
[‡]Barefoot Networks
[§]NYU

latency. When combined with the stateful and flexible packet processing capabilities of a programmable switch (such as Barefoot's Tofino and Cavium's Xpliant), we can customize the calendar queue abstraction to realize a broad class of scheduling algorithms that capture both physical and logical notions of time.

We demonstrate the power and flexibility of the calendar queue abstraction using three case studies. First, we use a calendar queue to perform deadline-aware scheduling of aggregate flows (or co-flows) and further use the same underlying calendar queue to implement fair queuing for the background traffic. The programmable switch pipeline performs the accounting operations required to keep track of the slack available for a packet and computes its rank as it traverses a network path. In our second case study, we use calendar queues to implement fair queueing and its variants that can tolerate a limited amount of burstiness, thereby providing a configurable balance between fairness and burst-friendliness. Here, the programmable switch pipeline maintains the flow state to perform the accounting operations required for scheduling. For our third case study, we realize pFabric and its variant that can provide a configurable amount of starvation prevention. We use a programmable pipeline to ensure the in-order transmission of packets even as the switch attributes higher priorities to later packets transmitted within a flow. We use a small-scale testbed comprising of Barefoot Tofino switches and a custom event-driven simulator to evaluate the benefits of these different scheduling algorithms.

2 Background

In this section, we discuss background material related to reconfigurable switches, the structure of the traffic manager (which is the same for both fixed-function and today's reconfigurable switches), and prior work on programmable packet scheduling.

2.1 Reconfigurable Switches

In this work, we assume that this programmable scheduling is used in conjunction with a reconfigurable switch, such as the Reconfigurable Match Table (RMT) model described in [8, 9]. A reconfigurable switch can operate at terabits/s by providing a restricted match+action (M+A) processing model: match on arbitrary packet header fields and perform simple packet processing actions. When a packet arrives at the switch, relevant headers fields are extracted via a user-defined parser and passed into a pipeline of user-defined M+A stages. Each stage matches on a subset of extracted headers and performs simple processing primitives or actions on any header. After traversing the ingress pipeline stages, packets are deposited in one of the multiple queues, typically 32–64, associated with the egress port for future transmission. On transmission, the packet goes through a similar egress pipeline undergoing further modifications.

In addition to a programmable parser and pipeline stages, these switches provide several hardware features to implement more complex use-cases: (1) a limited amount of stateful memory, such as counters and meters, which can be read and updated to maintain state across packets, (2) computation primitives, such as simple arithmetic operations and hashing, which can perform a limited amount of processing on header fields and data retrieved from stateful memory, and (3) the ability to recirculate or generate special datapath packets using *timers* that can be used to modify Traffic Manager status as well as synchronize the ingress and egress pipelines. Further, switch metadata, such as queue lengths, congestion status, and bytes transmitted, can also be used in packet processing. The complete packet processing, including parsing and match+action stages, can be configured using a high-level language, such as P4 [8]. A number of such switches, e.g., Cavium XPliant [11], Barefoot Tofino [6] and Intel Flexpipe [20], are available today.

A single pipeline's throughput is limited by the clock frequency achievable using today's transistor technology (typically about 1 GHz). To scale to higher packet-processing rates, which is required for switches with aggregate switch capacity in the Tbit/s range, the switch consists of multiple identical pipelines where the programmable stateful memory is local to each pipeline.

2.2 Traffic Manager

We now briefly describe the architecture of the traffic manager (TM) on merchant-silicon switches (e.g., Barefoot's Tofino and Broadcom's Trident series). The TM is responsible for two tasks: (1) buffering packets when more than one input port is trying to send packets to the same output port simultaneously, and (2) scheduling packets at each output port when the link attached to the port is ready to accept another packet.

Buffering: The TM is organized as a fixed number of first-in, first-out (FIFO) queues per output port. The ingress pipeline of the switch is responsible for determining both the FIFO queue and the output port that the packet should go to. Once the packet exits the ingress pipeline, the TM checks if there is sufficient space in the packet buffer to admit the new packet. Once the packet has been admitted to the buffer, it can not be dropped later because dropping a previously enqueued packet requires additional memory accesses to the packet buffer, which is expensive at line rate.

Scheduling: Packets are eventually dequeued from the buffer when the link attached to an output port goes idle and requests a new packet. During dequeue, the TM has to pick a particular FIFO at that output port, remove the earliest packet at that queue, and transmit it. The TM uses a combination of factors to determine which queue to dequeue from. First, each queue has a priority; queues with higher priority are strictly preferred to those with lower priorities. Next, within a priority level, the queues are scheduled in weighted round-round robin order (using an algorithm like DRR [26]). Lastly, each queue can be limited to a maximum rate and is paused and removed from consideration for scheduling if it has exceeded this rate over some time interval. Queues can also be paused because of a PFC [16] pause frame from a downstream switch.

To perform buffering and scheduling, the TM maintains a per-queue priority level, a per-queue pause status flag, and counters that track buffer occupancy on a per-input-port, per-PFC-class, per-output-port, and per-output-FIFO basis. These are required both to decide when and whether to admit packets and which queues to schedule. Because the status flag and counters support limited operations (e.g., toggling or increment/decrement), they can be implemented very efficiently in hardware, allowing simultaneous access from multiple ingress and egress switch pipelines in a single clock cycle (unlike state within the pipeline that can only be accessed once per clock cycle). It is important to note that these hardware mechanisms appear in traffic managers in both fixed-function and programmable switches.

2.3 Programmable Scheduling

The discussion above focused on a fixed-function TM that supports a small menu of scheduling algorithms (typically priorities, weighted round-robin, and traffic shaping). Recent proposals for programmable scheduling [2, 19, 23, 27–29] propose additional switch hardware in the TM to make the scheduling decision programmable, assuming the existence of a programmable ingress and egress switch pipeline like RMT. Of the proposals for programmable scheduling, we describe the PIFO work because it targets a switch similar to our paper, and it is representative of the hardware considerations associated with programmable scheduling. We defer a detailed comparison of both expressiveness and feasibility with prior work to later sections.

PIFOs enable programmable scheduling by using a programmable priority queue to express custom scheduling algorithms. Some external computation (either on the end host or a programmable switch's ingress pipeline) sets a rank for the packet. This rank determines the packet's order in the priority queue. By writing different programs to compute different kinds of packet ranks (e.g., deadlines or virtual times), different scheduling algorithms can be expressed using PIFOs.

While PIFOs are flexible, they have two shortcomings. PIFOs not only require the development of new hardware blocks that can scale with line rate (as we discuss below), but they are also limited given their support for a finite priority range (as we discuss in the next section). The scaling challenge arises out of needing to maintain a priority queue. The PIFO paper assumes that ranks within flows are naturally in strictly increasing order (i.e., flows are FIFOs), requiring the switch to only find the minimum rank among the head packets across all flows. While this reduces the sorting/ordering requirement of PIFO, sorting the total number of flows in the buffer is still challenging. The PIFO work provides a custom hardware primitive, the flow scheduler, which maintains a sorted array of a few thousand flows and can process tens of flows per output port across about 64 output ports on a single pipeline for an aggregate throughput of 640 Gbit/s. Scaling this primitive to higher speeds and a multi-pipeline switch can be challenging. Thus, a key goal of our programmable scheduling proposal is that it should be realized without increasing the temporal and spatial complexity of existing TM implementations.

3 Packet Scheduling using Programmable Calendar Queues

In this section, we begin by describing the Calendar Queue concept as introduced by Randy Brown in 1988 [10]. We then consider the abstraction of a Programmable Calendar Queue that combines the calendar queue scheduling mechanism with programmable packet processing pipelines. This combination allows for flexibility and extensibility, and we show how we can instantiate different variants of Programmable Calendar Queues to emulate different scheduling algorithms that appear in the literature.

3.1 Motivating Calendar Queues

Calendar Queues: The Calendar Queue was first introduced for organizing the pending event set in a discrete event simulation. It is a type of priority queue implementation that has low insertion and removal costs for certain priority distributions. Calendar Queues (CQs) are analogous to desk calendars used for storing future events for the next year in an ordered manner. A CQ consists of an array of buckets or queues, each of which stores events for a particular *day* in sorted order. Events can be scheduled for a future date by inserting the event in the bucket corresponding to the date. At any point, events are dequeued and processed from the current day in sorted order. Once all events are processed from the current day, we stop processing events for the current day and move onto the next day. The emptied bucket is then used to store tasks that need to be performed a year from now.

Drawbacks of existing priority queueing schemes: Prior work has made the observation that, scheduling algorithms make two decisions: in what order packets should be scheduled (in the case of work-conserving algorithms) or when they should be scheduled (in the case of non-work-conserving algorithms). For most scheduling algorithms, these decisions can be made at packet enqueue time. Comparison-based fine-grained priority queueing schemes, such as pHeap [7] or PIFO [28], can realize some of these algorithms by computing an immutable *rank* for a packet at packet enqueue time and dequeuing packets in increasing rank order. Eiffel [23] further observes that packet ranks often have a specific range (which can be expressed as integers) and that a large number of packets share the same rank. These characteristics make a bucket-based priority queue an efficient and feasible solution for implementing various scheduling algorithms.

We observe that many scheduling algorithms cannot be realized using fine-grained priority queuing schemes if the

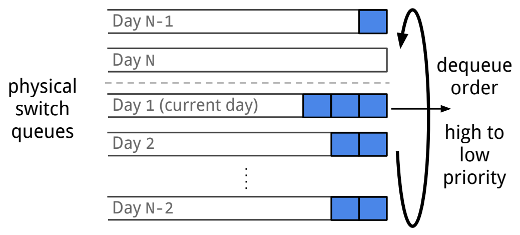


Figure 1: Example of a Programmable Calendar Queue.

computed rank needs to fall within a finite range. Consider the example of fair queuing (as in WFQ or STFQ), where for each arriving packet, a finishing round is computed based on the current round number and the finishing round of the previous packet in that packet's flow. Packets are then transmitted in order of increasing finishing round numbers. Further, the algorithm periodically increases the current round number. One could attempt to realize fair queuing using a fine-grained priority queuing scheme by mapping a packet's finishing round number to an immutable rank. Since the ranks of buffered packets cannot be changed, the mapping function needs to be monotonic, i.e., it needs to map higher finishing rounds to higher ranks. The mapping function would then exhaust any finite range of ranks, and the switch would then not be able to attribute a meaningful rank for incoming packets.

Similarly, in the case of the earliest deadline first (EDF), each packet in a flow is associated with a wall-clock deadline, and packets need to be scheduled in increasing order of deadlines. If one were to compute the rank of a packet as a monotonic function of the packet's deadline, the switch would exhaust the rank space as the wall-clock time progresses.

It is worth noting that, when the switch has no buffered packets, the mapping function could execute a "reset" and start reusing lower ranks. However, an implementation cannot assume that the switch would ever enter such a state, let alone periodically (i.e., within a bounded period of time before the rank space is exhausted).

Utility of Calendar Queues: We propose to use the Calendar Queues abstraction as a mechanism to realize scheduling algorithms such as EDF and fair queuing. A CQ is an attractive option for implementing these algorithms as it allows for implicit and en-masse escalation of the priorities of buffered packets when the CQ moves from one day to another. For instance, when a CQ completes processing the events for Day k and performs a *rotation* to Day $k+1$, it implicitly increases the priority of all days except Day k, which now occupies the lowest priority in the priority range. This rotation mechanism allows scheduling algorithms to escalate the priorities of buffered packets with time (as is the case with EDF and fair queuing) and reuse emptied buckets for incoming packets with low priority.

3.2 Programmable Calendar Queues (PCQs)

We now describe Programmable Calendar Queues in the context of reconfigurable switches. The programmable packet processing pipelines on these switches allow us to customize not only the rank computation but also the CQ rotation process. Just like a calendar has 365 days, we assume our Calendar Queue abstraction has a fixed number of buckets or FIFO queues, say N, each of which stores packets scheduled for next N periods (see Figure 1). Any network scheduling algorithm using CQs must then make the following key decisions. First, the scheduling algorithm must decide how far in the future the incoming packet should be scheduled, i.e., choose a future period from [0, N-1] to enqueue the packet into. This is similar to rank computation in PIFO. Second, it must periodically decide when and how to advance time, i.e., decide when a period is over and move onto the next period. This stops the enqueueing of packets in the current period and allows the reuse of the corresponding queue resource for the period that is *N* periods into the future. Third, when the CQ advances to the next period, the pipeline state has to be suitably modified to ensure the appropriate computation of ranks for incoming packets.

The advancing of time can be done using a physical clock, i.e., the CQ moves onto the next queue after a fixed time interval periodically; we call this a *Physical* Calendar Queue. Alternatively, the CQ can advance to the next queue whenever the current queue is empty, i.e., it happens logically depending on metrics such as bytes sent or number of communication rounds; we call this a *Logical* Calendar Queue. A Physical Calendar Queue lets us implement both work-conserving schemes, such as EDF, and non-work-conserving schemes, such as Leaky Bucket Filter, Jitter-EDD, and Stop-and-Go, whereas a Logical Calendar Queue can implement work-conserving schemes, such as LSTF, WFQ, and SRPT.

We now list the interface methods exposed to the packet processing pipelines that enable these forms of customization.

- **CQ.enqueue(n)**: Used by the ingress pipeline to schedule the current packet *n* periods into the future.
- **CQ.dequeue()**: Used by the egress pipeline to obtain a buffered packet, if any, for the current period.
- **CQ.rotate()**: Used by the pipelines to advance the CQ so that it can start transmitting packets for the next period.

We observe that PCQs have certain properties that allow for efficient implementations. (In Section 3.4, we describe how to realize this abstraction in hardware.) When individual CQ periods are mapped to separate physical switch queues, a CQ scheduler needs to maintain state only at the granularity of switch queues (e.g., the queue corresponding to the current period). The scheduler does not require expensive sorting or comparisons to determine packet transmission order. More importantly, a CQ rotation involves a deterministic and predictable transition from one switch queue to another at the end of each period. This transition can be realized either using data-plane primitives in upcoming reconfigurable hardware (as we discuss in Section 3.4) or through the switch's control plane (as is the case with our prototype).

3.3 Programmable Scheduling using PCQs

We now show how various scheduling algorithms can be realized using Calendar Queues in conjunction with a programmable packet processing pipeline. We describe three different algorithms, each of which differs in the way it utilizes CQs. First, an approximate variant of WFQ that uses a *Logical* CQ. Next, we implement approximate EDF using LSTF scheduling that uses a work-conserving *Physical* CQ. Finally, we realize a Leaky Bucket Filter that utilizes a non-work-conserving *Physical* CQ.

3.3.1 Weighted Fair Queueing

Weighted Fair Queueing (WFQ) scheduling achieves max-min fair allocation among flows traversing a link by emulating a bit-by-bit round-robin scheme where each active flow transmits a single bit of data each round. This emulation is realized at packet granularity by assigning each incoming packet a departure *round number* based on the current round number and the total bytes sent by the flow. All buffered packets are dequeued in order of increasing departure round numbers.

Packet State
```
 weight : Packet flow’s weight
```
Switch State
```
 bytes[f] : Number of bytes sent by flow f
 round    : Current round number
 BpR      : Bytes sent per round for each flow
```
Rank Computation & Enqueueing
```
 bytes[f] = max(bytes[f], round * BpR * weight)
 n = (bytes[f] + pkt.size) / (BpR * weight) - round
 CQ.enqueue(n)
```
Queue Rotation
```
 if CQ.dequeue() is null
   CQ.rotate()
   round = round + 1
```

Figure 2: WFQ implementation using a Logical CQ.

We implement WFQ using Logical CQs closely following the round number approximation described in [25]. We use coarse-grain rounds that are only incremented after all active flows have transmitted a configurable quantum of bytes. The rank computation is done in such a way that each fair queuing round is mapped to a day (queue) in the Calendar Queue and, whenever a day finishes (i.e., the queue is drained completely), the round number is incremented by one. The complete switch state and computation required is shown in Figure 2. Note that this is an approximation of the WFQ algorithm where the round numbers are not as precise or faithful to the original algorithm, and there can be situations in which packets are transmitted in an *unfair* order. However, this *unfairness* has an upper-bound and is controlled by the BpR variable in the rank computation. As we show later in the evaluation, this approach closely approximates ideal fair queueing.

3.3.2 Earliest Deadline First

In Earliest Deadline First (EDF) scheduling, each packet from a flow is assigned a deadline or expected time of reception. At each network hop, the packet with the closest deadline is transmitted first. We implement EDF using Least Slack Time First scheduling, where each packet carries a *slack* value of the time remaining till its deadline. The slack is initialized to `deadline-arrivalTime` at the source and updated at each hop along the way (i.e., each switch subtracts the time spent at the hop from the slack). The implementation uses a Physical CQ, as shown in Figure 3, which we describe next.

Packet State
```
 slack : Initialize to flow_deadline - arrival_time
```
Switch State
```
 dT      : Time interval of each queue
 delta   : Skew between ideal and measured time
 lastRot : Timestamp of last rotation
```
Rank Computation & Enqueueing
```
 n = (slack - delta + (currentTime - lastRot)) / dT
 CQ.enqueue(n)
```
Queue Rotation
```
 if CQ.dequeue() is null
   CQ.rotate()
   delta = delta + (dT - (currentTime - lastRot))
   lastRot = currentTime
```

Figure 3: EDF using a work-conserving Physical CQ.

We choose a fixed time interval for each *day* or queue for our Physical CQ, say dT. Packets with an effective slack of $0 - dT$ are assigned to queue 1, slack of $dT - 2 \cdot dT$ are assigned to queue 2, and so on. This assignment ensures that packets with closer deadlines are prioritized. Queue rotation occurs when the current queue becomes empty. Since we can spend a longer or shorter time than dT in any queue depending on the traffic pattern, we require some additional state to ensure that new packets are inserted in the correct queue with respect to the deadlines of already enqueued packets. The `delta` variable keeps track of how far ahead the CQ is compared to the ideal time. If we spend less than dT for a queue, *delta* increases, and if we spend more than dT, it decreases. The *delta* is then incorporated in the rank computation and is reset to 0 whenever there are zero buffered packets. Note that the programmable switch pipeline allows us to perform not only the rank computation but also decide when to perform the CQ rotation and how to update switch state after a rotation.

3.3.3 Leaky Bucket Filter

A Leaky Bucket Filter (LBF) is a non-work-conserving scheduling algorithm that rate limits a flow to a specified bandwidth and a maximum backlog buffer size. An LBF can be realized using a Physical Calendar Queue by storing a fixed quantum of bytes per flow in each queue and rotating

Packet State

```
rate : Output rate limit
size : Maximum bucket size
```

Switch State

```
dT       : Time interval size of each queue
bytes[f] : Bytes sent by flow f
round    : Current round number
```

Rank Computation & Enqueueing

```
bytes[f] = max(bytes[f], round * rate * dT)
n = bytes[f] / (rate * dT) - round
if n > size / (rate * dT)
  drop packet
else
  CQ.enqueue(n)
```

Queue Rotation

```
if dT time has elapsed
  CQ.rotate()
  round = round + 1
```

Figure 4: A Leaky Bucket Filter using a non-work-conserving Physical CQ.

queues at fixed time intervals, very similar to the WFQ example discussed earlier. However, we do not dequeue packets from the next queue even if the current queue is empty, which gives us the desired non-work-conserving behavior. The byte quantum depends on the rate limit set for the flow and the configured time interval dT of each queue. If the number of enqueued bytes for a flow exceeds the bucket size, we simply drop the packet. This scheme lets us realize multiple filters using the same underlying CQ, as shown in Figure 4.

We assume the configured `rate` and `size` parameters for the filter are in the packet header, but they can be stored at the switch as well. For each flow, we keep track of bytes sent so far and compute the queue id by dividing it with the quantum, which is the configured filter rate times the queue interval dT. We assume that the cumulative rate of all flows does not exceed the line rate at the switch; if that happens, all flows will be slowed down in proportion to their configured rates equally.

3.4 Implementing PCQs in Hardware

We now describe how Calendar Queues can be implemented on programmable switches. We assume an RMT model switch (as described in Section 2) with an ingress pipeline, followed by the traffic manager, which maintains multiple packet queues, and finally an egress pipeline. Implementing a CQ in this model is non-trivial because the packet enqueue decision (i.e., which queue to insert the packet into) is made in the ingress pipeline, but the queue status (i.e., occupancy, depth) is available in the egress pipeline after the packet traverses the traffic manager. Since these modules are implemented as separate hardware blocks, we need to *synchronize* state among them to achieve the CQ abstraction.

We can realize CQs on programmable switches using mutable switch state, multiple FIFO queues, the ability to create and recirculate packets selectively, and the ability to pause/resume queues or alter queue priorities directly in the data plane. All these capabilities are either already available in today's programmable switches except for the data-plane-based queue pausing, resuming, and priority updating. However, we confirmed with experts in switching ASIC design that adding such a capability is relatively straightforward because doing so doesn't change the order of temporal or spatial complexity of existing TM implementations. Existing TMs already support several per-packet metadata to expose controllable features or statistics. Moreover, PFC already requires a similar queue pausing/resuming capability triggered by certain protocol messages in the data plane anyway. This new feature exposes a similar functionality only in a programmatic way by exposing such knobs to the programmable pipeline. Moreover, even in the absence of data-plane support for priority changes, we can still approximate this functionality using the control plane (as we do in our testbed).

Implementation Overview: We first provide a high-level description of our scheme. Each *period* in the CQ is mapped to a single FIFO queue within a set of queues associated with the outgoing port. The ingress pipeline computes which *period* or queue each incoming packet is enqueued into. We assume a TM that allows the pipeline to enqueue incoming packets into any of the available FIFO queues. At any given time, the queue settings satisfy the following properties: (a) The queue corresponding to the current period has the highest priority level, so that its packets can be transmitted immediately. We refer to this queue as the head queue. (b) The queue corresponding to the next period has a lower priority level and is active/unpaused. (c) All other queues corresponding to future periods are at the lowest priority level and are paused. This specific configuration has two desirable properties. First, since the next period's queue is also active, the switch can start transmitting packets from the next period's queue as soon as the head queue becomes empty. Second, when we perform a CQ rotation, we need to perform only a small number of changes to queue priorities and active statuses.

The CQ cycles or rotates through available queues one at a time, making each queue the head queue. The rotation is triggered when the head queue is empty (in case of a logical CQ) or the CQ time interval has elapsed (in case of a physical CQ). When a rotation happens, we need to make sure all packets from the head queue are drained completely and that it is empty before changing its priority to lowest. Once the priority is set, the head queue can be reused to store packets for future periods.

Implementation Details: We break down the implementation of CQs into the following three steps. Appendix A provides additional details.

Step 1 - Initiate Rotation: This step detects when a rotation needs to happen and initiates the rotation by informing the ingress pipeline using a recirculation packet. In the case of a logical CQ, we initiate rotation when the head queue is empty. This can be detected in two ways. First, in some

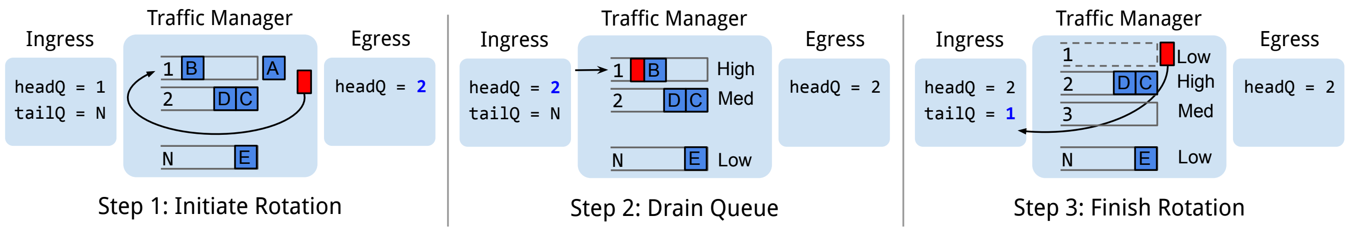


Figure 5: Step 1, the egress pipeline tries to dequeue from the current headQ, and if it is the last packet, it initiates rotation by creating and recirculating a `rotate` (red packet) to the ingress pipeline. Step 2, the ingress pipeline on receiving this packet, updates the headQ and enqueues a `marker` packet as the last packet into the old headQ. Finally in Step 3, when the egress pipeline receives the `marker` packet, it updates the queue priority and notifies the ingress pipeline that it is safe to reuse the queue for future packets by updating the tailQ.

programmable switches, the traffic manager metadata could provide the egress pipeline information regarding the depth of the queue from which the packet was dequeued. If it is zero, this is the last packet from the head queue, and we initiate rotation. Second, we can check the queue id from which the packet was dequeued to infer whether the head queue is empty. Since the successor head queue is also unpaused with a lower priority, a packet dequeued from it implies the head queue is empty. We use the latter method in our prototype as it imposes fewer requirements on the traffic manager metadata available to the switch pipeline. In the case of a physical CQ, the rotation happens at fixed time intervals and is configured through timers or packet generators. When a rotation begins, we recirculate a special `rotate` packet to the ingress pipeline so that it stops enqueuing packets in the head queue and begins draining it, which happens in Step 2.

Step 2 - Drain Queue: This step ensures that the head queue is completely drained, and no more packets are enqueued into it till the rotation finishes. On receiving the `rotate` recirculation packet, the ingress pipeline advances the head queue pointer, essentially stopping any new packets from being enqueued into it. But, there could still be some packets in the pipeline currently making their way into the head queue. To make sure these are transmitted in the right order, the ingress enqueues a special `marker` packet into the head queue after updating the head of the calendar queue. This packet is the last packet to be enqueued into the head queue, and its arrival at the egress pipeline means the queue is completely drained, and we can proceed with finishing the rotation described in Step 3.

Step 3 - Finish Rotation: The `marker` packet is recirculated back to the ingress pipeline, and this informs the ingress pipeline that it is safe to reuse the queue for future periods. The ingress changes the priority of the just emptied queue to lowest and also pauses it, essentially pushing the queue to the end of the CQ. The ingress also unpauses the queue associated with the next period to ensure that there are no transmission stalls after the current period ends. The queue configuration change can be achieved in two ways depending on the underlying hardware support. The `marker` packet can be pushed up to the control plane CPU, which can alter the queue configurations using traffic manager APIs. This approach incurs a latency overhead before the drained queue can be used for packets associated with future periods. Alternatively, if the hardware supports priority change in the datapath, the processing of the `marker` packet with the appropriate metadata tags affects the configuration change almost immediately.

We now briefly highlight some of the attributes of PCQ that aid in efficient hardware realization. First, CQs maintain state at the granularity of physical switch queues instead of individual packets or flows. Second, at any given point in time, there is a designated head queue that is responsible for providing the packets that are to be transmitted. Third, the rotation operation involves changing just the metadata of queue and that too of at most three queues. This combination of factors allows us to bolt-on the PCQ abstraction on to a traditional TM.

3.5 Analysis and Extensions

We now analyze our PCQ abstraction and compare it to both fine-grained priority queuing schemes and an ideal Calendar Queue along different dimensions. We also provide an extension that expands the scheduling capability of the PCQ.

Expressiveness: From a theoretical perspective, a static priority mechanism (e.g., PIFO) with infinite priority levels is equivalent to a Calendar Queue with infinite buckets, and most scheduling algorithms can be expressed in both these hypothetical schemes. However, a practical PIFO has finite priority levels, and a practical CQ has finite buckets, which affects the feasibility and the fidelity of scheduling algorithms. An algorithm can be implemented using PIFOs if all packets throughout the "lifetime" of the algorithm have ranks strictly in the priority queue range. This is true for algorithms like pFabric where packet rank is solely a function of flow size, but not true for WFQ or EDF where packet ranks are computed based on a round number or current time, which is monotonically increasing. As discussed earlier, the priority-level space will roll over eventually, and the ordering of enqueued packets will be violated. On the other hand, our realization of PCQs requires that the enqueued packets' ranks at any instant fit within the available buckets or queues, which makes it challenging to implement algorithms that require both a large packet rank range as well as high fidelity in distinguishing between the packet ranks; we can extend the range of a CQ by bucketing several packet ranks together, but that introduces approximations which we discuss next.

Approximations: There are two sources of approximations that arise in a PCQ. First, inversions within a FIFO queue. The original Calendar Queue [10] maintains events in a single bucket in sorted order, whereas we simply use a FIFO queue. This can lead to inversions if multiple ranks are assigned to the same bucket, i.e., a higher priority packet is scheduled after a lower priority packet. This presents a feasibility vs. accuracy trade-off for the scheduling algorithm, and if the bucket intervals are chosen carefully, the approximation is acceptable as we show later in the evaluation. It is worth noting that one could borrow some of the mechanisms from the SP-PIFO work [2] to reduce rank inversions, but we leave that to future work. Second, the PCQ imposes a limit on the range of the CQ. Since the number of FIFO queues is limited, there is a possibility that packets will arrive with a rank beyond the range of the CQ. One can theoretically increase the bucket size to include a larger priority schedule such packets that are very far in the future. However, this will lead to an increase in inversions and reduce the accuracy of the priority queuing mechanism. Another option is to store overflowing packets into a separate queue and recirculate them into their appropriate queue when they get close to their service time. Furthermore, the range of a CQ can be significantly increased by employing a hierarchical structure, and we describe this next.

Hierarchical Calendar Queues: One way to extend the range of a CQ is to employ a hierarchical structure among the available FIFO queues, similar to hierarchical timing wheels, at the cost of recirculating some data packets. To create a 2-level hierarchical calendar queue (HCQ), we split the N FIFO queues into two groups of sizes n_1 and n_2, respectively. The two groups run independent calendar queues CQ_1 and CQ_2 on top of them, however with different bucket intervals: CQ_1 having an interval of one time period and CQ_2 having an interval of n_1 time periods, as shown in Figure 6. The idea is that a single queue of CQ_2 has an interval equivalent to the full rotation of all n_1 queues of CQ_1. A packet with a scheduled time between 1 to n_1 is inserted into the appropriate queue in CQ_1, packets with time between n_1+1 to $2\times n_1$ are inserted into the first queue of CQ_2, packets with $2\times n_1+1$ to $3\times n_1$ are inserted into the second queue of CQ_2, and so on. This approach provides a total range of $n_1\times n_2$ time periods, whereas just using a single CQ over N queues would give a range of just n_1+n_2.

However, this comes at the cost of recirculating any data packet that is enqueued in CQ_2. When a full rotation of all n_1 queues in CQ_1 finishes, all packets from the head queue of CQ_2 are recirculated and deposited into appropriate queues in CQ_1. Note that this approach is still significantly better

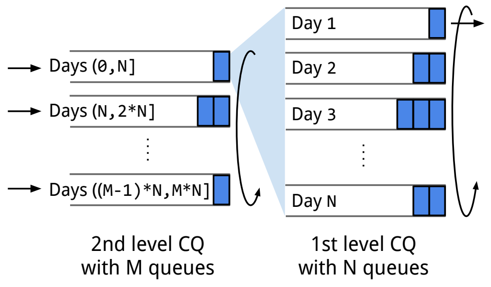


Figure 6: Example of a 2-level Hierarchical CQ. Packets are enqueued into the higher level CQ if they are too far into the future and recirculated into the finer level CQ periodically.

than the approach described in [10], where packets scheduled too far in the future are simply enqueued in the scheduled queue *modulo N*, and recirculated if the scheduled time has not arrived; Brown's scheme can recirculate a packet multiple times, whereas an HCQ recirculates a packet only once leading to more efficient use of bandwidth. Implementing HCQs also requires storing and managing extra state for both CQs and more complex computations when determining the destination queue for a packet. With 32 FIFO queues, a 2-level HCQ can be implemented with 16×16 queues to achieve a reach of 256 time periods or a 3-level HCQ with $16\times8\times8$ queues with a total reach of 1024, which is significantly larger than 32.

Limitations: Similar to PIFOs, CQs compute the enqueue rank only on packet arrival. Therefore, the relative order of already buffered packets cannot be changed after enqueuing. This limitation prevents CQs from realizing mechanisms such as pFabric's starvation prevention technique [4], where the dequeue order of multiple previously received packets changes on an enqueue; a later packet within a flow would signal that there are fewer remaining bytes within the flow and, thereby, increase the priority of the flow's previously enqueued packets. CQs do allow us to realize other, arguably stronger, forms of starvation prevention, as we will see in Section 4.5. Another limitation of CQs is that we can schedule packets only in the future. If the computed rank is before the current CQ time, the resulting packet schedule will be different from the desired ordering. Essentially, this means that CQs cannot correctly order packets that have ranks in the past. One could address this limitation by not immediately reusing a queue for a future period as soon as we perform a CQ rotation and allowing some number of queues from the past to be active. Finally, CQs have an upper limit on the range of ranks they can enqueue at a time. If a scheduling algorithm requires a large range, it cannot be implemented accurately using CQs; the hierarchical scheme outlined above can increase a CQ's range, but it comes with an approximation cost.

4 Evaluation

We evaluate the practical feasibility, expressiveness, and performance of Calendar Queues by implementing them on a

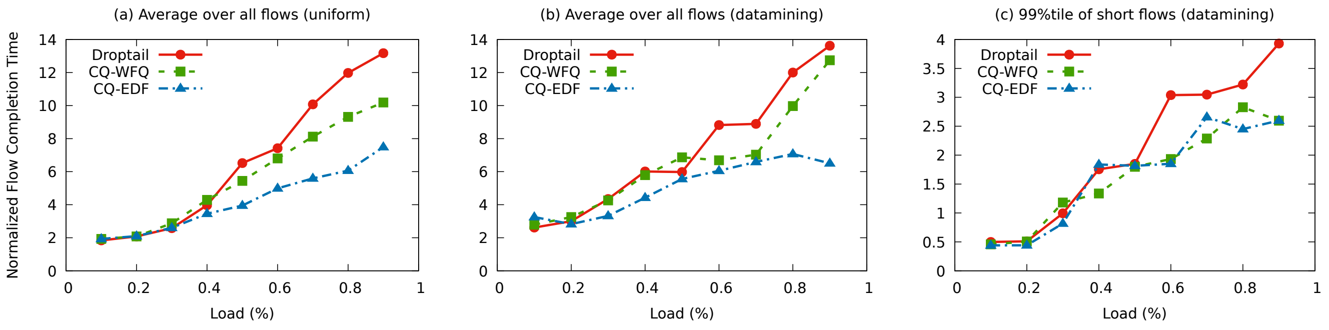


Figure 7: Average and tail latencies for both synthetic and datamining workloads with WFQ and EDF policies implemented using Calendar Queues on top of Barefoot Tofino switch in the hardware testbed.

programmable Barefoot Tofino switch and realizing two classical scheduling algorithms using CQs. Next, using large scale packet-level simulations, we demonstrate the flexibility of the Calendar Queue abstraction with three case studies. First, we instantiate a physical calendar queue that performs deadline-aware scheduling of both aggregate flows (or coflows) and individual flows. Second, we instantiate a logical calendar queue that implements a variant of fair-queueing that can tolerate a limited amount of burstiness, thereby providing a configurable balance between fairness and flow completion time for short flows. Finally, we implement a variant of pFabric that prevents long flows from starving by gradually increasing the priority of all enqueued packets. None of these scheduling algorithms can be realized using traditional priority queuing schemes such as PIFO.

4.1 Hardware Prototype Implementation

We implement and evaluate programmable calendar queues on the Barefoot Tofino 100BF-32X switch. As the current Tofino switch does not support updating a queue's priority on the datapath, we implement CQs using a combination of in-built packet generator, packet re-circulation, and control plane operations to drain packets in the correct order.

First, we use the in-built packet generator on the switch to periodically generate probe packets and detect when a queue rotation needs to be performed. The egress pipeline tracks the current head of the CQ, and when a packet is dequeued from the next queue (signifying the current queue is empty), it re-circulates the probe packet back to the ingress to initiate a rotation. Next, the ingress pipeline updates the current head of the CQ and enqueues the probe packet into the queue being rotated out to drain it fully, and no more packets are inserted into it. When the egress pipeline receives the probe packet again, it is safe to update the priority of the drained queue, and we achieve this by setting a flag in the egress pipeline. Finally, the control CPU polls on this flag variable, and when set, it makes an API call to update the queue's priority and notifies the ingress pipeline that it is safe to use this queue to store the future packets.

Testbed and Workload We implement two scheduling algorithms, WFQ [14] and EDF [18] using Calendar Queues and compare them against standard FIFO droptail scheduling in a 2-level fat-tree topology, consisting of 2 ToR switches, 2 aggregation switches, and 4 servers by using loopback links with ingress-port based virtualization to divide the 32 physical ports into multiple switches. All links in the network are 40Gbps with 80µs end-to-end latency. Each server opens 80 concurrent long-running connections to other servers and generates flows according to a Poisson process at a rate configured to achieve the desired network load. We tested a synthetic workload that draws flow sizes at uniform with a max size of 12.5 MB and the data mining workload from [4].

Performance Figure 7 shows the average and 99th percentile latencies. Across both workloads, we can see the WFQ and EDF implementations performing better than simple droptail queues. The difference is more significant at higher network load when queues build up at the switch due to bursty arrivals. This is when the prioritization and correct scheduling of packets leads to a visible difference in FCTs. However, the important point here is that we were able to realize these scheduling policies at a line-rate of 40 Gbps without being limited by the number of flows. We further measured the extra physical switch resources consumed by our implementation of WFQ and EDF using CQs, and report them in Appendix B.

4.2 Packet-level Simulations

We study our use-cases in a large-scale cluster deployment using an event-driven, packet-level simulator. We extend the mptcp-htsim simulator [21] to implement Calendar Queues and several other comparison schemes. The simulation consists of 256 servers connected in a 3-level fat-tree topology consisting of 8 ToR switches, 8 aggregation switches, and 4 core switches. Each ToR switch is connected to 32 servers using 10 Gbps links, and all other switches are connected to each other using 40 Gbps links. Next, we describe each case-study and evaluate them.

4.3 Use Case 1: Coflow scheduling using Least Slack Time First (LSTF) scheduling

Distributed applications running inside datacenters generate network traffic patterns that require optimizing the perfor-

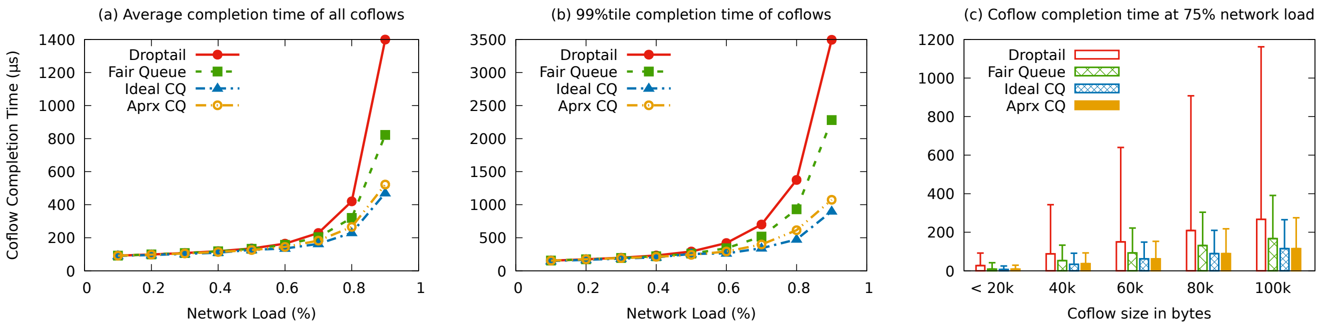


Figure 8: Coflow completion time when running a mix of background and coflow traffic with coflows prioritized over background flows. (a) average CCT for all coflows, (b) 99th percentile CCT for all coflows, and (c) average and 99th percentile (using error bar) for various coflow size buckets at 75% network load.

mance on a collection of flows [12], called coflows, rather than individual flows, e.g., partition-aggregate or bulk synchronous programming tasks such as multi-get Memcached queries and MapReduce jobs. The performance of such applications depends on the last finishing flow among the collection and prior work [1] has shown that near-optimal performance can be achieved by ordering coflows using a Shortest Remaining Processing Time (SRPT) first mechanism and ensuring that any packet from any flow of a coflow X ordered before coflow Y is transmitted before any packet from coflow Y.

We implement the above approach using LSTF scheduling on top of Calendar Queues to optimize coflow completion times (CCT). However, instead of using the complex BSSI algorithm in [1], which decides priorities based on other coflows in the system, we choose a much simpler heuristic to order coflows as they arrive. We compute a deadline for the whole coflow, assuming the largest sub-flow in the coflow is the bottleneck and will be the last to finish. Therefore, the deadline is simply the largest sub-flow size divided by endhost link speed. All we need to do is assign this deadline to all packets of all flows in the coflow and ensure that packets with the earliest deadline are transmitted first at each switch.

We calculate each packet's *slack* as the time remaining until the deadline of its corresponding coflow. The slack is initialized in the packet header at the endhost, and as the packet traverses the network, each switch enqueues the packet in the Calendar Queue based on this slack value. The higher the slack value, the farther in future the packet is scheduled for transmission. On departure, the switch deducts the time spent at the switch from the slack and updates the packet header. As a result, critical flows with lower slack values and closer deadlines are dynamically prioritized over non-critical flows with larger slack values and farther deadlines. Note that this scheme could have been implemented using an exact priority queue (such as PIFO) with absolute deadlines embedded in the packet header, but that would require clocks to be synchronized. More importantly, the switch would run out of priority levels eventually and would not be able to enforce deadlines. We, therefore, implement the scheme using LSTF on top of Calendar Queues.

We measure the performance of the above coflow scheduling mechanism using event-driven simulations and compare it with the following queueing schemes:

- **Droptail**: Traditional switch with a single FIFO queue that drops packets from the tail when full.
- **Fair Queue**: Bit-by-bit round-robin algorithm from [14] that achieves max-min fairness.
- **Ideal Calendar Queue**: A CQ with *infinite* buckets that also transmits packets in sorted order within each bucket.
- **Approx Calendar Queue**: Our implementation of CQs that uses 32 FIFO queues and 10µs round interval.

In all the above schemes, we use the same end-host flow control protocol, DCTCP, with the additional embedding of *slack* value based on the coflow deadline.

Workload and Performance Metric We use a mix of background traffic, which is the enterprise workload in [3] and a synthetic coflow workload derived from a Facebook trace [1]. Coflows, on average, have ten sub-flows and a total size of 100 KB. The ratio of background traffic to coflow traffic is 3:1. Flows and coflows arrive according to a Poisson process at randomly and independently chosen source-destination server pairs with an arrival rate chosen to achieve the desired level of utilization in the aggregation-core switch links. We evaluate the performance in terms of flow completion time (FCT) or, in case of coflows, the coflow completion time (CCT), which is the maximum FCT of comprising sub-flows and report both mean and 99th percentile numbers.

Since we have a mix of background flows and coflows, we must decide how they co-exist together and are scheduled inside the network. One trivial way is to treat background traffic as a separate class with a lower priority than coflow traffic. This configuration is evaluated in Setup 1. Another option made possible by CQs is to treat background traffic as fair-queued and coflow traffic as deadline-aware using the same underlying CQ to schedule packets belonging to both classes. This demonstrates the flexibility of CQs in realizing multiple scheduling policies at once, and we evaluate this configuration in Setup 2.

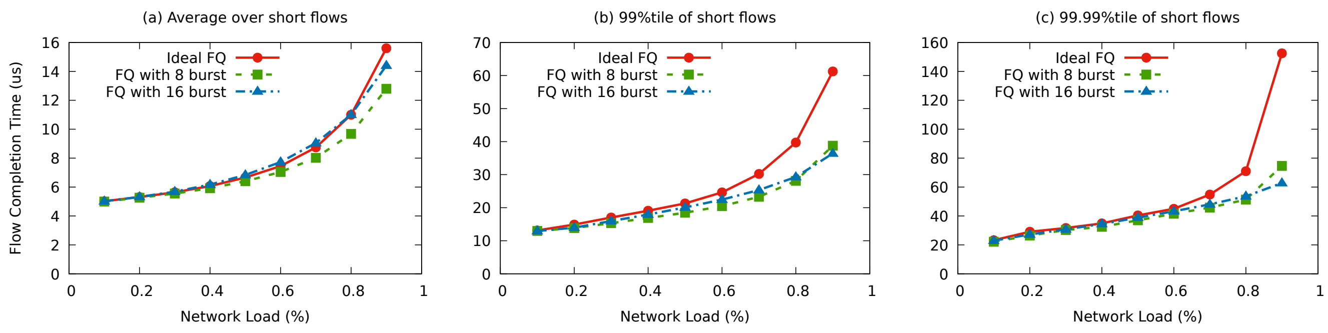


Figure 9: Average and percentile flow completion times for short flows with varying network load and permissible burst size.

Setup 1: Coflows have priority over background traffic. We treat background flows as a different traffic class with lower priority than coflows by using a separate queue with strictly lower priority. As a result, the background traffic performance is not affected by the coflow scheduling policy, and we report coflow completion times in Figure 8. The average CCT improves by up to 3x and 99th percentile CCT by 5x at high network loads. This improvement is both because we are emulating an SRPT policy as well as because we de-prioritize shorter sub-flows within a coflow over other more critical flows. Both droptail and fair-queuing finish short flows within a coflow quickly, although they have a significant *slack* till the deadline. Moreover, our CQ implementation is able to accurately emulate an ideal calendar queue mechanism using a limited number of FIFO queues, and the gap between ideal CQ and our CQ is fairly negligible.

We present the results for Setup 2 in Appendix C.

4.4 Use Case 2: Weighted Fair Queueing with Burst Allowance

First, we implement Fair Queueing using calendar queues as described in Section 3.3.1, which emulates a round-robin scheme wherein each active flow transmits a fixed number of bytes each round (*BpR*). The departure round computed for each packet is mapped to a future day (or queue) in the CQ, which transmits the packets in the correct order. This scheme has been shown to emulate Fair Queuing accurately [25] and while it implements Start Time Fair Queueing at a coarse granularity, it is often desirable to allow a burst of packets from a single flow to be transmitted back-to-back to achieve a better latency for short flows and improve tail latency [13]. This only affects fairness at very short timescales while maintaining fairness at larger timescales.

We modify WFQ to allow short bursts to go through using a simple modification to the enqueuing logic at the ingress. In addition to maintaining a byte counter per flow and computing a packet's round number as byte counter divided by *BpR*, we maintain a permitted *burst size*. While our original WFQ implementation allows a flow to send at most *BpR* bytes per round, the burst-friendly variant lets a flow enqueue up to a fraction of available burst size into a single round, allowing it to exceed its fair share allocation temporarily.

More precisely, instead of computing the round number as `R = bytes[f]/BpR`, where `bytes[f]` is the amount of bytes enqueued by flow `f`, we incorporate burst size into the calculation as follows,

```
R = bytes[f]/max(BpR, BurstSize - bytes[f])
```

where `BurstSize` is the configured permissible burst size. This essentially lets `BurstSize/2` bytes to be enqueued in the current round, `BurstSize/4` in the next round, and so on, as show in Figure 10. Any enqueued bytes exceeding the burst size are assigned the same round number as before.

We implement this burst-friendly fair-queuing scheme with configured burst sizes of 8 and 16 packets (denoted by FQ-8 and FQ-16), along with ideal fair-queueing on top of our Calendar Queues and measure the impact on flow completion times using simulations. We use the same 3-level fat-tree topology and enterprise workload for this use case.

Figure 9 shows the flow completion times of short flows as we increase the network load. At higher network loads, allowing a burst of bytes to go through leads to up to 2-3x reduction in higher percentile latencies. Figure 11 breaks down the latency improvement across different flow size buckets, and it confirms that short flows in the region of burst size show the most improvement. More importantly, larger flows are unaffected by this temporary burst allowance.

4.5 Use Case 3: pFabric with Starvation Prevention

pFabric [4] is a transport layer designed to provide near-optimal flow completion time by essentially emulating Shortest Remaining Processing Time (SRPT) first scheduling at each switch. Each flow packet carries a single number that encodes its priority – in this case, it is set to the remaining flow size when the packet is transmitted. All switches forward the packet with the shortest remaining flow size at any given time. This simple scheme can be implemented using a static fine-grained priority scheme such as PIFO, as it does not require the gradual priority escalation. However, this leads to potential starvation of long flows, which are always deprioritized compared to shorter flows, making it impractical to run in real environments. This is shown in Table 1, where we simulated the same 3-level fat-tree topology and ran the enterprise workload at 80% network load. As flow size increases,

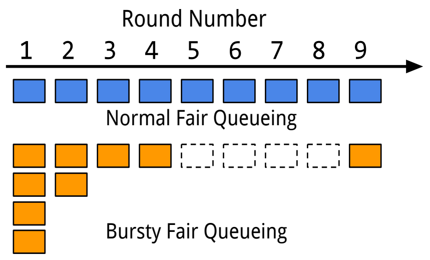


Figure 10: Packet enqueuing behavior of normal fair queueing vs bursty fair queueing.

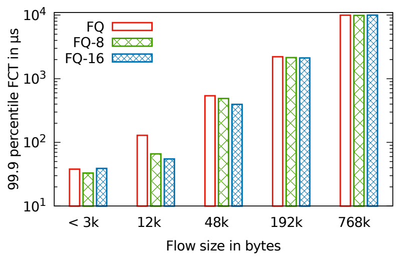


Figure 11: 99.9 percentile latency for various flow size buckets at 90% network load in micro-seconds.

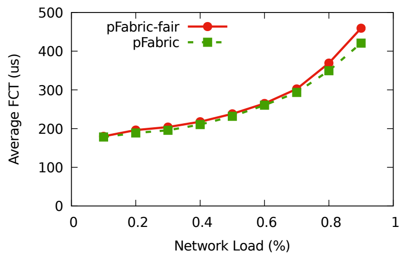


Figure 12: Average FCT for pFabric and pFabric-fair for enterprise workload in our testbed.

Flowsize	10k	100k	1M	10M	100M
pFabric	679	651	670	524	265
pFabric-fair	650	642	638	543	442

Table 1: Average bandwidth in MBps achieved by flows in various bucket sizes.

the average transmission rate decreases resulting in reduced bandwidth available for longer flows.

If we were to think of fair queuing and pFabric as ends of the spectrum, calendar queues would provide us with options in the middle. We implement a fairer version of pFabric, called pFabric-fair, by slightly altering the enqueuing mechanism. A packet with k bytes remaining in the flow is enqueued $f(k)$ periods into the future, where $f(k)$ is a log function. Thus, higher rounds are exponentially bigger, and the CQ can accommodate large flow sizes. Whenever the current head queue is empty, we rotate to the next queue, which ensures that low priority packets from larger flows are not permanently starved, merely deprioritized at enqueue time, and their priorities increase with time. However, we need some additional state to ensure that we do not enqueue a later packet from a flow ahead of the flow's previously received packets, which we achieve by keeping track of the highest queue for each flow.

Figure 12 shows the average and 99th percentile FCT for all flows with varying network loads for pFabric and pFabric-fair. Although pFabric-fair has a slightly higher average FCT at higher network loads, it also provides higher bandwidth to large-sized flows, preventing them from starving.

5 Related Work

Packet schedulers available in switching hardware today are fixed function, supporting specific primitives such as strict priority, rate limits, round-robin fairness, although a vast number of richer scheduling algorithms that provide stronger guarantees exist in the literature, such as WFQ [14], pFabric [4], STFQ [15], SRPT [24], EDF [18].

Several recent proposals aim to provide a programmable packet scheduler that can implement these scheduling algorithms while operating at line rate of terabits per second, such as PIFO [28], PIEO [27], and SP-PIFO [2]. All of these proposals provide the abstraction of *static* and *finite* priority levels, which we argue is insufficient to implement several scheduling algorithms that require monotonically escalating priorities. pHeap [7], PIFO and PIEO provide fine-grained priority levels, which makes it challenging for them to scale to large packet buffers and multi-pipeline switches. SP-PIFO is similar to Calendar Queues as it also provides coarse-grained priority levels using only FIFO queues, and can scale to current line-rate switches. SP-PIFO dynamically adjusts the priority range of individual queues by changing queueing thresholds, which can be explored further in the context of Calendar Queues as well.

Another chain of work proposes efficient packet scheduling in software such as Carousel [22], Loom [29], and Eiffel [23]. These approaches rely on timing wheel data structures or bucketed integer priority queue-like data structures for efficient operation. Calendar Queue borrows ideas from similar data structures while targeting switching hardware that can support today's large buffer and multi-pipeline routers.

6 Conclusion

We propose a flexible packet scheduler designed for line-rate hardware switches, called Programmable Calendar Queues, that enables the efficient realization of several classical scheduling algorithms. It relies on the observation that most algorithms require both prioritization and implicit escalation of a packet's priority. We show how they can be implemented efficiently on today's programmable switches by dynamically changing the priority of queues using either dataplane primitives or control-plane operations. We demonstrate that PCQs can be used to realize interesting variants of LSTF, Fair Queueing, and pFabric to provide stronger delay guarantees, burst-friendly fairness, and starvation-free prioritization of short flows, respectively.

Acknowledgments

We would like to thank the anonymous NSDI reviewers and our shepherd Rachit Agarwal for their valuable feedback. This research was partially supported by NSF Grant CNS-1714508 and Futurewei.

References

[1] Agarwal, S., Rajakrishnan, S., Narayan, A., Agarwal, R., Shmoys, D., and Vahdat, A. Sincronia: Near-optimal network design for coflows. In *Proceedings of the ACM SIGCOMM Conference* (2018).

[2] Alcoz, A. G., Dietmüller, A., and Vanbever, L. SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, 2020).

[3] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Lam, V. T., Matus, F., Pan, R., Yadav, N., and Varghese, G. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the ACM SIGCOMM Conference* (2014).

[4] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM Conference* (2013).

[5] Bai, W., Chen, L., Chen, K., Han, D., Tian, C., and Wang, H. Information-agnostic flow scheduling for commodity data centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA, 2015).

[6] Barefoot Networks. Tofino Programmable Switch. `https://www.barefootnetworks.com/technology/`.

[7] Bhagwan, R., and Lin, B. Design of a High-speed Packet Switch with Fine-grained Quality-of-Service Guarantees. In *Proceedings of the IEEE International Conference on Communications* (June 2000), vol. 3, pp. 1430–1434 vol.3.

[8] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (July 2014).

[9] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM Conference* (2013), pp. 99–110.

[10] Brown, R. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31* (1988), 1220–1227.

[11] Cavium. XPliant Ethernet switch product family. `http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.

[12] Chowdhury, M., Zhong, Y., and Stoica, I. Efficient coflow scheduling with varys. In *Proceedings of the ACM SIGCOMM Conference* (2014), SIGCOMM '14, pp. 443–454.

[13] Clark, D. D., Shenker, S., and Zhang, L. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *Proceedings on the ACM SIGCOMM Conference* (1992).

[14] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. In *Proceedings on the ACM SIGCOMM Conference* (1989).

[15] Goyal, P., Vin, H. M., and Chen, H. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of the ACM SIGCOMM Conference* (1996), SIGCOMM '96, pp. 157–168.

[16] IEEE. Priority based flow control. *IEEE 802.11Qbb* (2011).

[17] Leung, J. Y.-T. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica 4*, 1-4 (1989), 209.

[18] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20*, 1 (Jan. 1973), 46–61.

[19] Mittal, R., Agarwal, R., Ratnasamy, S., and Shenker, S. Universal packet scheduling. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016).

[20] Ozdag, R. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf`.

[21] Raiciu, C. MPTCP htsim simulator. `http://nrg.cs.ucl.ac.uk/mptcp/implementation.html`.

[22] Saeed, A., Dukkipati, N., Valancius, V., The Lam, V., Contavalli, C., and Vahdat, A. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the ACM SIGCOMM Conference* (2017), SIGCOMM '17, pp. 404–417.

[23] Saeed, A., Zhao, Y., Dukkipati, N., Zegura, E., Ammar, M., Harras, K., and Vahdat, A. Eiffel: Efficient and flexible software packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), pp. 17–32.

[24] Schrage, L. E., and Miller, L. W. The queue m/g/1 with the shortest remaining processing time discipline. *Oper. Res. 14*, 4 (Aug. 1966), 670–684.

[25] Sharma, N. K., Liu, M., Atreya, K., and Krishnamurthy, A. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), pp. 1–16.

[26] Shreedhar, M., and Varghese, G. Efficient fair queueing using deficit round robin. In *Proceedings on the ACM SIGCOMM Conference* (1995).

[27] Shrivastav, V. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM SIGCOMM Conference* (2019), SIGCOMM '19, pp. 367–379.

[28] Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S.-T., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., and McKeown, N. Programmable packet scheduling at line rate. In *Proceedings of the ACM SIGCOMM Conference* (2016).

[29] Stephens, B., Akella, A., and Swift, M. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, 2019), pp. 33–46.

A Detailed Implementation Notes for Realizing PCQs in Hardware

We now present the traffic manager API and how it is invoked by the different steps involved in transitioning a CQ from one period to another.

Traffic Manager API

- `tm_enqueue(packet, queue)`: Enqueues a packet in the given queue.
- `tm_pause(queue)`: Stop or pause queue from transmitting any packets until unpause is called on the same queue.
- `tm_unpause(queue)`: Resume queue, allowing it to send out packets until pause is called.
- `tm_setPriority(queue, p)`: Set priority of queue to `p` (one of different levels supported by the TM)
- `tm_dequeue()` (called from egress): Returns a packet from the highest priority unpaused queue, along with the queue id from which it was dequeued.

The exact details of enqueue/dequeue and rotation are described in the pseudocode below. In addition, we store the following state at ingress and egress to keep track of Calendar Queue status and perform queue rotations. The special recirculation packets also contain metadata regarding which queue is being rotated out.

Ingress State

```
currQ: queueId currently at the head of the CQ
prevQ: queueId that was just rotated out and is draining
nextQ: queueId that will be unpaused next
```

Egress State

```
currQ: queueId at the head of the CQ (egress)
```

Packet Enqueue Function

```
packet_enqueue(pkt, X) (called from ingress)
  // According to desired scheduling algorithm.
  x = compute_rank(pkt)
  // Enqueue packet in queue (currQ + x) % N
  tm_enqueue(pkt, (currQ + x) % N)

  // process initiate rotate packet
  if packet == rotate:
     ingress.prevQ = ingress.currQ
     ingress.currQ = ingress.nextQ
     ingress.nextQ = (ingress.nextQ + 1) % N
     tm_enqueue(marker, ingress.prevQ)
     tm_setPriority(prevQ, High)
     tm_setPriority(currQ, Medium)

  // process marker packet
  if packet == marker:
     tm_pause(marker.queueId)
     tm_unpause(nextQ)
     tm_setPriority(nextQ, Low)
```

Packet Dequeue Function

```
packet_dequeue() (called from egress)
  // Returns the highest priority packet
  pkt, queueId = tm_dequeue()

  // Need to initiate rotation
  if queueId != egress.currQ:
     create and circulate a rotate packet on egress.currQ
     egress.currQ++

  // Normal packet dequeue
  if queueId == egress.currQ:
     perform normal packet processing

  if pkt == marker:
     recirculate marker back to ingress
```

Note that, if no more packets remain to be transmitted, then rotate packet is never sent out and currQ remains the same. This avoids unnecessary rotations when the link traffic is less than the link bandwidth.

B Resource Overhead of Implementing CQs

Table 2 shows the additional overhead of implementing CQs along with various scheduling policies, as reported by the P4 compiler. First, since we were able to compile CQs directly onto the Tofino hardware, we can support an arbitrary number of flows at the configured line-rate of 40 Gbps, and are not scale limited in any way. We do require some additional state that is proportional to the number of CQs instantiated across all ports and the number of queues in each CQ. Each scheduling policy also keeps extra state for rank computation, which takes extra resources, e.g., keeping flow byte counters for WFQ results in 50% increase in SRAM usage.

Resource	Baseline	CQ w/ EDF	CQ w/ WFQ	CQ w/ EDF+WFQ
Pkt Header Vector	356	356	356	356
Pipeline Stages	9	9	12	12
Match Crossbar	50	54	63	68
Hash Bits	113	124	140	150
SRAM	27	29	46	48
TCAM	2	2	2	2
ALU Instruction	11	12	13	14

Table 2: Summary of resource usage for a Calendar Queue implementation with 32 physical queues on top of P4 switch.

C Scheduling Deadline-aware and Background Traffic using the same CQ

In Setup 2 described in Section 4.3, we use the same underlying Calendar Queue to schedule background flows as fair-queued traffic and coflows as deadline or slack based traffic. For a background flow packet, which is fair-queued, we compute its departure queue based on bytes enqueued by the flow using the WFQ implementation described in Section 3.3.1. For a deadline-aware coflow packet, we calculate

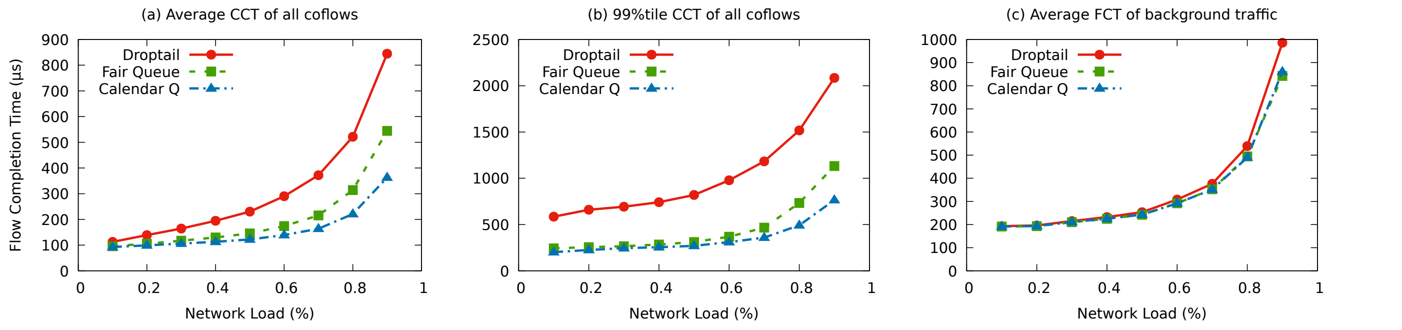


Figure 13: CCT and FCT when scheduling background traffic as fair-queued and coflow traffic as deadline-aware using the same Calendar Queue.

its departure queue using the slack inside the packet header by dividing it with the configured bucket interval as described in Section 3.3.2. We can control the relative priority of background vs. coflow traffic by changing the bucket interval. A higher bucket interval will accommodate more bytes from deadline traffic compared to fair-queued traffic. We use a default value of 10µs as the bucket interval and 1 MSS as the bytes quantum per round for fair queueing.

Figure 13 shows the coflow completion times and FCT of background flows in this setup. The average CCT shows up to 2.5x and 1.5x improvement compared to droptail and fair queuing, respectively. This benefit again comes from the fact that we are able to schedule shorter coflows before longer coflows, as well as de-prioritize shorter sub-flows within a coflow over other more critical sub-flows. The 99th percentile shows a similar improvement of 3x over droptail queues. Moreover, the average FCT of background flows stays roughly the same and is unaffected by the coflow scheduling being done by the Calendar Queue.

Contra: A Programmable System for Performance-aware Routing

Kuo-Feng Hsu
Rice University

Ryan Beckett
Microsoft Research

Ang Chen
Rice University

Jennifer Rexford
Princeton University

Praveen Tammana
Princeton University

David Walker
Princeton University

Abstract

We present Contra, a system for performance-aware routing that can adapt to traffic changes at hardware speeds. While point solutions exist for a fixed topology (e.g., a Fattree) with a fixed routing policy (e.g., use least utilized paths), Contra can operate seamlessly over any network topology and a wide variety of sophisticated routing policies. Users of Contra write network-wide policies that rank network paths given their current performance. A compiler then analyzes such policies in conjunction with the network topology and decomposes them into switch-local P4 programs, which collectively implement a new, specialized distance-vector protocol. This protocol generates compact probes that traverse the network, gathering path metrics to optimize for the user policy dynamically. Switches respond to changing network conditions by routing flowlets along the best policy-compliant paths. Our experiments show that Contra scales to large networks, and that in terms of flow completion times, it is competitive with hand-crafted systems that have been customized for specific topologies and policies.

1 Introduction

Configuring a network to achieve a diverse range of objectives, such as routing constraints (*e.g.*, traffic should go through a series of middleboxes), and traffic engineering (*e.g.*, minimize latency and maximize throughput), is a challenging task. To handle this complexity, one approach has been to use SDN solutions, which have a centralized point for management [25, 26]. However, centralized controllers are inherently too slow to respond to fine-grained traffic changes, such as short traffic bursts. In fact, even the software control planes locally on the switches are often limited in their ability to select new routes fast enough.

Recent work has developed load-balancing mechanisms that operate entirely in the data plane to enable real-time adaptation [11, 30]. By making use of fine-grained performance information on hardware timescales, these systems can deliver considerable performance benefits over static load-balancing mechanisms like ECMP. Unfortunately, existing systems, such as Conga [11] and Hula [30], are point solutions that only work under specific assumptions about the network topology, routing constraints, and performance objectives—they only support a "least utilized shortest path" policy on a data center topology. It is not obvious how to adapt them for other kinds of topologies or policies.

In this paper, we describe Contra, a general and programmable system for performance-aware routing. Network operators configure Contra by describing the network topology as well as a high-level policy that defines routing constraints and performance objectives. Contra then generates P4 programs for switches in the network, which execute in a fully distributed fashion. Collectively, they implement a specialized version of a distance-vector protocol that forwards traffic based on routing constraints and optimizes for the user-defined performance objectives. This protocol operates by generating periodic probes that traverse policy-compliant paths and collect user-defined performance metrics. Switches analyze the incoming probes and rank paths in real time, storing the current best next hop to reach any given destination. Since the programs run in the data plane, switches can react to performance changes quickly. Overall, Contra is designed to achieve the following objectives:

- General – operates over a wide range of policies
- Reusable – works correctly for any topology
- Distributed – does not require central coordination
- Responsive – adapts to changing metrics quickly
- Implementable – on today's programmable data planes
- Policy-compliant – packets only use allowed paths
- Loop-free – mitigates persistent/transient loops
- Optimal – converges to best paths under stable metrics
- Stable – mitigates oscillation under changing metrics
- Efficient – avoids undue traffic and switch overhead
- Ordered – limits out-of-order packet delivery

To achieve these objectives, we need to address several challenges. First, to operate over arbitrary topologies, Contra requires new techniques to search the set of possible paths for optimal routes. State-of-the-art solutions, such as Conga [11] and Hula [30], assume a tree-based data center topology, which makes exploring possible paths, avoiding forwarding loops, and finding optimal routes straightforward. Second, link and path metrics can change constantly, which may

Objective	Key idea(s)	Section(s)
General	Language for performance-aware routing Policies as path-ranking functions	2
Reusable	Policy analyzed jointly with topology	4.1
Distributed Responsive & Implementable	Synthesis of data-plane routing protocol Periodic probes to collect path metrics Implemented in P4	4.1–4.3
Policy-compliant	Probes and packets carry policy states Switches keep track of state transitions	4.1–4.3
Loop-free	Monotonicity analysis Probes carry version numbers Early loop breaking for flowlets	2, 5.1, 5.5
Optimal Stable & Efficient	Isotonicity analysis Limit the frequency of probes Failure detection and metric expiration	2, 5.2, 5.4
Ordered	Policy-aware flowlet switching	5.3

Figure 1: Key ideas in Contra.

cause unsynchronized views at different switches. Making forwarding decisions based on inconsistent views may lead to forwarding loops or paths that violate the routing policy. Third, a naïve solution that constantly changes routes can cause transient or even persistent chaos. We draw inspirations from wireless network routing [16, 38, 39], and design mechanisms that leverage programmable data planes to address this. Finally, we develop policy-aware flowlet switching, which routes flowlets to mitigate out-of-order packet delivery while ensuring policy compliance.

Summary. We make several contributions in the design of Contra, and Figure 1 summarizes the key ideas.

- We define a new programming abstraction that views policies as path-ranking functions, and generalizes existing languages by allowing operators to specify path constraints and dynamic metrics simultaneously.
- We design a new configurable, performance-aware, distance-vector routing protocol.
- We develop compilation algorithms that generate switch-local P4 programs that implement a particular configuration of the protocol based on user policy.
- We have built a system prototype, and conducted thorough experiments to demonstrate that Contra is competitive with state-of-the-art systems that are customized for a specific topology and routing policy.

Non-goals. There has been abundant recent research on efficient load-balancing strategies, especially in data centers. The goal of this work is not to outperform such strategies in the contexts for which they have been manually optimized. Rather, our goal is to facilitate the deployment of such techniques on a much broader set of networks and with a broader collection of optimization criteria, and to do so without asking network operators to take the time, or acquire the expertise necessary, to write "assembly-level" P4 programs.

2 Policy language

Contra includes a high-level language that can express a wide range of user policies, which are functions that rank network paths. Our compiler then ensures that switches always use the best policy-compliant paths. Users can combine regular expressions, which express hard constraints on the allowed paths, with performance metrics to express dynamic preferences. As a concrete example, consider the following policy:

```
minimize( if A .* then path.util else path.lat )
```

It first classifies paths using a regular expression (`A .*`), and then based on the classification, it defines the rank to be either path utilization or latency. Each node will separately choose its best paths according to this function. So node A will always choose the least utilized path, while all other nodes will select the path with the lowest latency.

The Contra language can also capture static policies in existing systems that are not related to performance. For instance, FatTire [40] uses regular expressions to classify legal and illegal paths (though it says nothing about the *performance* of such paths). To route packets through a waypoint `W`, a FatTire policy would be `(.* W .*)`, which allows any path through `W` but no other paths. Contra can represent this by mapping all legal paths to 0 and illegal paths to ∞:

```
minimize( if .* W .* then 0 else ∞ )
```

This policy will ensure that every node always selects a path through `W` if one exists in the network, and drops traffic otherwise; no path is preferred to a path with rank ∞.

As another example, Propane [14] allows users to write policies about failover preferences. A Propane policy `(A B D) >> (A C D)` indicates a preference for sending traffic through path `A B D` and only using `A C D` if the first path is not available (*e.g.*, a link has failed). In Contra, we can achieve the same effect by ranking paths statically as below.

```
minimize( if A B D then 0 else if A C D then 1 else ∞ )
```

In Contra, it is also possible to rank paths based on multiple metrics. For example, suppose we prefer that A reaches D via B instead of via C, and we also prefer shorter, less utilized paths. This can be achieved by lexicographically ranking paths, *e.g.*, prefer paths through B first, then shortest paths, and finally, least utilized paths.

```
minimize( if A .* B .* D then (0, path.len, path.util)
      else if A .* C .* D then (1, path.len, path.util)
      else ∞ )
```

Ranking paths using regular expressions defines strict, inviolate preferences; however, operators may have softer constraints based on path performance: *e.g.*, one path may be preferred up to a point, but if the utilization is too high then some traffic should be shunted along another path instead. For example, to prefer least-utilized paths when the network load is light (utilization of the path is less than 80%), even if those paths are long, but to prefer shortest paths when network load is heavy (and hence to save bandwidth globally), one might use the following policy.

```
minimize( if path.util < .8
            then (1, 0, path.util)
            else (2, path.len, path.util) )
```

Policy

$$\begin{array}{lll} pol & ::= & \mathsf{minimize}(e) \qquad\qquad\qquad \textit{optimization} \end{array}$$

Expressions

$$\begin{array}{llll} e & ::= & n & \textit{constant numeric rank} \\ & \mid & \infty & \textit{infinite rank} \\ & \mid & \mathsf{path.attr} & \textit{path attribute} \\ & \mid & e_1 \circ e_2 & \textit{binary operation} \\ & \mid & \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 & \textit{if statement} \\ & \mid & (e_1, \ldots, e_n) & \textit{tuple} \end{array}$$

Boolean Tests

$$b \ ::= \ r \mid e_1 \leq e_2 \mid \mathsf{not}\ b \mid b_1\ \mathsf{or}\ b_2 \mid b_1\ \mathsf{and}\ b_2$$

Regular Paths

$$r \ ::= \ \mathsf{node\_id} \mid . \mid r_1 + r_2 \mid r_1\ r_2 \mid r^*$$

Figure 2: Syntax for Contra policies.

Finally, to steer traffic towards or away from particular links, one may add or subtract weights. For instance, the following policy demonstrates how to add weight to costly links AB and CD while otherwise using simple shortest paths.

```
minimize( (if .* AB .* then 10 else 0) +
          (if .* CD .* then 20 else 0) + path.len )
```

Figure 2 presents the full language syntax, and Table 1 presents selected policy examples taken from the literature. The key novelty of the language is that it can capture many of the *static* conditions expressed by earlier work such as Fat-Tire [40] or NetKAT [13] as well as the *relative* preferences of Propane [14], and yet it also augment such policies with *dynamic* preferences based on current network conditions.

Policy	Implementation
P1. Shortest path routing [24]	path.len
P2. Minimum utilization [30]	path.util
P3. Widest shortest paths [32]	(path.len, path.util)
P4. Waypointing [13]	if .*(F_1+F_2).* then path.util else ∞
P5. Link preference [14]	if .*XY.* then path.util else ∞
P6. Weighted link [19]	(if .*XY.* then 10 else 0) + path.len
P7. Source-local preference [12]	if X.* then path.util else path.lat
P8. Congestion-aware routing [27]	if path.util < .8 then (1, 0, path.util) else (2, path.len, path.util)

Table 1: Selected Contra policies.

Policy analysis and guarantees. Contra requires user policies to be *monotonic* (metrics do not improve for longer paths) and *isotonic* (switches have consistent preferences). If a policy is non-isotonic (*e.g.*, P8), Contra will attempt to decompose it into multiple isotonic subpolicies that can be processed separately. Contra can do this for many conditional policies (*e.g.*, P8), but it will not always succeed, e.g., for "shortest widest paths"; see Appendix A for more discussion. These algebraic constraints guarantee that when metrics are stable, new flows will be sent along globally optimal paths [22]. Our system also guarantees that hard constraints expressed by regular expressions are never violated. Under changing metrics, when switches make distributed decisions based on their local views, routes may be suboptimal [11].

3 Selected Challenges

Contra addresses three key challenges. To illustrate these challenges, we first describe a simple strawman solution designed for a specific topology (data center networks) and specific policy (use least utilized paths). Consider the simple leaf-spine topology in Figure 3(a), where switch S wants to send traffic to switch D over the least-utilized path:

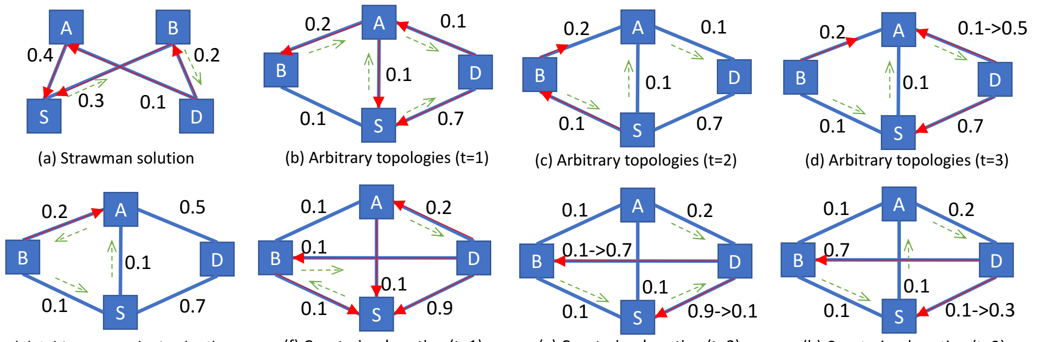
```
minimize( if S.*D then path.util else ∞)
```

One strawman solution is to use a distance-vector protocol, where each switch propagates link metrics (*i.e.*, utilization) to its neighbors via periodic probes, and builds up a local forwarding table of "best next hops" to reach other switches.

Concretely, at time 1, D sends two probes to A and B carrying utilizations u(A-D)=0.1 and u(B-D)=0.2, respectively. Upon receiving a probe, a spine switch updates its metric, and then disseminates the probe to its downstream neighbors. The updated probe metric is the maximum of a) the original probe metric, and b) the utilization of the inbound link from the switch's neighbor, so the probe always carries the utilization of the bottleneck link on its traversed path. For instance, when B receives the probe from D, it updates the utilization to 0.3, which is the maximum of a) the original probe metric, u(B-D)=0.2, and b) the utilization u(S-B)=0.3; when A receives the probe from D, it updates the utilization in the probe to be 0.4, which is the maximum of u(A-D)=0.1 and u(S-A)=0.4. At time 2, both A and B disseminate the updated probes to S. Now, S has received probes on both paths S-A-D (u=0.4) and S-B-D (u=0.3), and it chooses B as the best next hop to reach D due to its lower utilization. Changes in link metrics are then propagated by the next round of probes. In fact, this describes Hula [30], a state-of-the-art solution for utilization-aware routing in data centers.

Challenge #1: Arbitrary topologies. On a tree topology, simple mechanisms (*e.g.*, defining a set of "downstream" and "upstream" neighbors for each switch) suffice to explore paths and prevent forwarding loops [30], but on a non-hierarchical topology, it is insufficient.

Consider the sequence of events in Figures 3(b)-(e), where S prefers the least-utilized path to D. Suppose that at time 1, D sends out probes to A and S, and A propagates D's probe to B and S, with the utilizations shown in Figure 3(b); now, both B and S prefer to reach D via A. At time 2, S propagates A's probe to B about S-A-D (u=0.1), so B changes its preference to go through S; B then propagates S's probe to A (u=0.2), but it gets delivered only at time 4. At time 3, u(A-D) increases to 0.5, which is discovered by a new periodic probe from D to A and S. From A's perspective, the best path to reach D is still A-D, except that now the utilization is 0.5 instead. At time 4, when B's (old) probe to A arrives with u=0.2, A mistakenly thinks that it should instead reach D via B, not knowing that A is itself on B's best path to reach D. As a result, a forwarding loop S-A-B-S would form, and it will *persist* as long as the link utilizations remain stable.



(a) Strawman solution (b) Arbitrary topologies (t=1) (c) Arbitrary topologies (t=2) (d) Arbitrary topologies (t=3)
(e) Arbitrary topologies (t=4) (f) Constrained routing (t=1) (g) Constrained routing (t=2) (h) Constrained routing (t=3)

Figure 3: Supporting sophisticated policies over arbitrary topologies is challenging. (Solid, red arrows represent probes, and dotted, green arrows represent packet forwarding. Links are labeled with performance metrics.)

It might seem that path-vector protocols would address this problem, where probes record their traversed paths, and switches avoid picking paths that involve themselves. However, the root cause of transient loops is the inconsistent views during network convergence; so transient loops can still form even with path-vector protocols [37]. Carrying the path traversed by the probe would also increase traffic overheads and the complexity for processing probes.

Solution. Our solution is inspired by DSDV [39] and a more recent proposal Babel [16], which were originally developed for wireless mesh networks. At a high level, switches assign version numbers to probes, so that they can identify and avoid using outdated probes. In addition, Contra uses flowlet switching [44] to pin traffic to particular paths and avoid out-of-order packet delivery. Still, because flowlet entries expire at different times, it is possible for transient loops to form on rare occasions. Contra quickly detects and breaks such loops by monitoring hop counts.

Challenge #2: Constrained routing. Supporting routing policies with path constraints leads to additional challenges. Consider the scenario in Figure 3(f), where the policy is not only to prefer least-utilized paths, but also that traffic should never first go through B and then A due to security concerns:

```
minimize(if .*B.*A.* then ∞ else path.util)
```

Under this policy, S can only send traffic to D via a) S-D, b) S-A-D, c) S-B-D, or d) S-A-B-D; initially, S prefers c) (u=0.1). Now consider the sequence of events shown in Figures 3(f)-(h). Suppose that at time 1, the traffic from S arrives at B. At time 2, the u(B-D) increases to 0.7, and u(S-D) decreases to 0.1, so B updates its best next hop (to reach D) to be S, preferring the path B-S-D. At time 3, B sends the traffic back to S, which already forms a loop. But things can get even worse: at time 3, u(S-D) increases to 0.3, so S changes its preference to be S-A-D (u=0.2). So the traffic has been forwarded along a path S-B-S-A-D, which not only contains a loop but also violates the intended policy.

Solution. Contra compiles the regular expression constraints in the user policy into automata, and intersects these automata with the network topology to obtain a *product graph* [45, 14], which specifies a probe and packet tagging scheme for each switch. Intuitively, tags represent states of the user-defined automata; by checking that probes and packets carry the right state when arriving at a switch, it is possible to enforce the global user policy in a distributed fashion. This tagging scheme guarantees that no packet ever deviates from a user's regular expression constraints in the policy.

Challenge #3: Custom performance metrics. Supporting custom performance metrics also introduces new challenges. As discussed earlier, a switch only propagates the probe with the *best* metric to its neighbors. However, such local decisions do not always give rise to globally optimal results, unless the policy is *isotonic* [22] (*i.e.*, roughly speaking, downstream nodes respect the preferences of upstream nodes). Unfortunately, some useful policies, such as some congestion-aware routing schemes, are not isotonic [27].

Solution. Contra analyzes the user policy to determine if it is isotonic. If not, Contra decomposes the non-isotonic policy into multiple isotonic subpolicies. Information about each subpolicy is propagated separately in different classes of probe and the best probe from each class is chosen locally. The classes are recombined and a route corresponding to the best current path is chosen only at a traffic source. Hence, if metrics are stable, then new flows will be sent along globally optimal paths. To avoid packet reordering due to unstable metrics, we follow Conga and Hula's strategy and use flowlet switching, which trades the fact that packets in pinned flowlets may follow suboptimal paths for stability.

4 Compilation: Stable metrics

The goal of the compiler is to generate a particular configuration of the Contra protocol that efficiently implements the desired policy in the data plane. We describe compilation in two phases. First, in this section, we describe an algorithm that operates *as if link metrics do not change*, so probes only need to be propagated once. The next section explains how this algorithm is extended to handle changing metrics.

Challenge. One key challenge during compilation involves

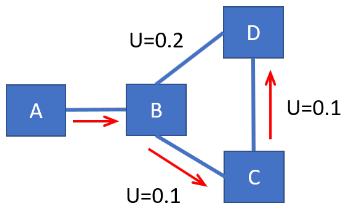


Figure 4: Naïve solutions may lead to suboptimal paths. Node `A` uses `ABCD` even though a better path `ABD` exists.

policies with conditional regular expression matches, such as `(if r then m1 else m2)`, because nodes may rank paths differently based on the branch of the conditional they use. In fact, conditional regular expression matches are one source of non-isotonicity: if every node selects the best next hop according to its own preferences alone, other nodes might wind up with suboptimal routes. For example, consider the following policy when applied to the topology in Figure 4:

```
minimize( if (A B D) then 0 else path.util)
```

In this example, `A` prefers path `ABD`, but `B` prefers the least utilized path `BCD`. The correct behavior in this scenario would be for `B` to carry `A`'s traffic along path `ABD` while simultaneously sending its own traffic along path `BCD`.

However, a naïve (and erroneous) implementation may disseminate probes along the paths `DB` and `DCB`[1] and ask `B` to decide which path is best. In this case, `B` would use the probe from `DCB` and discard the one from `DB`. However, if the latter probe is discarded, `A` will not receive information about its preferred route! To avoid this, another naïve solution would be to propagate probes along all possible paths in the network to avoid missing good paths. For instance, `B` might send every probe it receives to `A`. However, this would lead to far too many probes, as the number of paths in a graph may be exponential in the number of nodes.

Solution. Instead, for a conditional `(if r then m1 else m2)`, if one could determine the path with minimal metric `m1` that matches `r` using one probe, and separately determine the path with minimal metric `m2` that does not match `r` using another probe, then nodes could delay choosing their best path until both probes have been received and only then combine the information to make a decision. This is one concrete instance where Contra needs to decompose the non-isotonic policy (due to regular expressions) into multiple isotonic subpolicies. Contra achieves this by creating an efficient data structure that combines all regular expressions appearing in a policy with the network topology, and by sending separate probes for different regular expression matches.

4.1 Finding policy-compliant paths

Inspired by Merlin [45] and Propane [14], Contra constructs a data structure called a *product graph* (PG), which compactly represents all paths allowed by the policy.

[1]Recall that probes travel in the opposite direction to actual traffic.

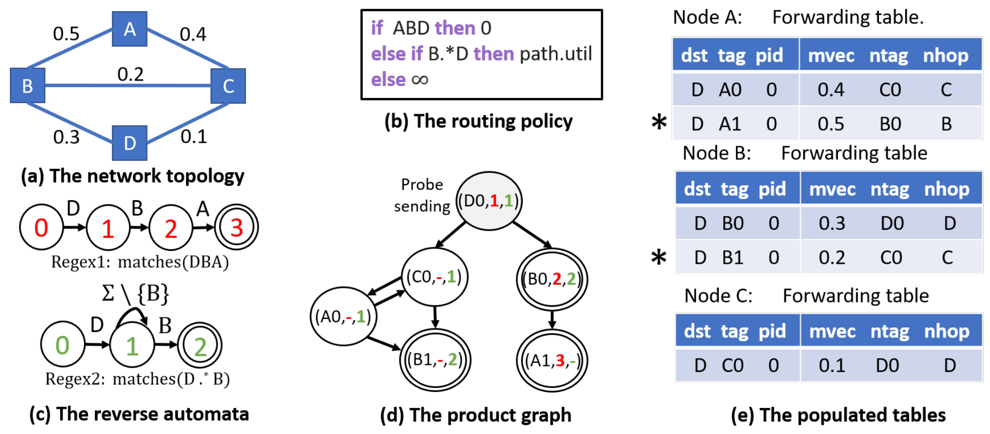
Policy automata. A policy's regular expressions define the different ways the shape of a path can affect its ranking. To process a policy, we first convert all such regular expressions into finite automata. Because probes disseminate information starting from the destination, but policies describe the direction of traffic that flows in the opposite direction, we actually construct an automaton for the reverse of each regular expression. Each automaton is a tuple $(\Sigma, Q_i, F_i, q_{0_i}, \sigma_i)$. Σ is the alphabet, where each character represents a switch ID in the network. Q_i is the set of states in automaton i. The initial state is q_{0_i}. F_i is the set of accepting / final states. $\sigma_i: Q_i \times \Sigma \to Q_i$ is the transition function. Consider the example policy in Figure 5(b), which a) allows A to reach D via the path A-B-D, b) allows B to reach D via any path with the least utilization, and c) disallows all other paths. The Contra compiler would generate the automata in Figure 5(c).

Network topology. The construction of the automata has not considered the actual network topology, so not all automaton transitions are legitimate. For instance, although the automaton for `D.*B` could in principle accept a sequence of transitions `D-A-B`, this sequence would never happen on the network shown in Figure 5(a), simply because `D` is not directly connected to `A`. Therefore, our compiler merges the topology with the automata and prunes invalid transitions.

Product graph (PG). If there are k automata (one for each regular expression used in the policy), then each state in the PG would have $k+1$ fields, $(X, s_1, \cdots, s_k)$, where the first field X is a topology location, and s_i is a state in the i-th automaton; there is a directed edge from $(X, s_1, \cdots, s_k)$ to $(X', s'_1, \cdots, s'_k)$, if a) $X - X'$ is a valid link on the topology, and b) for each automaton i, we have $\sigma_i(s_i, X') = s'_i$.

Concretely, in the PG in Figure 5(d), every edge represents both valid transitions on the two policy automata and a valid forwarding action on the topology. As examples, no edges exist from any `(D,*,*)` state to `(A,*,*)` state, because they have been eliminated due to topology constraints; also, there is a transition between node `D0` and `B0` because a) the topology connects `D` and `B`, and b) applying `B` to each automaton from state 1 leads to state 2. We use the symbol "–" to denote the special "garbage" state—the state from which there is no valid transition in an automaton.

Virtual nodes. We distinguish PG nodes ("virtual nodes") from topology locations ("physical nodes"). A physical node X may have multiple virtual nodes, because probes could arrive via different paths, and reach different automaton states as a result. For instance, the physical node `B` has two virtual nodes `(B0,-,2)` and `(B1,2,2)`; we have labeled their location fields as `B0`, `B1` to capture this, and we call them *tags*. If multiple virtual nodes exist, then probes must be duplicated to traverse paths that satisfy different constraints. For instance, `B` will receive a probe for `B0` representing a path on the way to matching regex `ABD`, and a second probe for `B1` representing a path on the way to matching regex `B.*D`.



Node A: Forwarding table.

	dst	tag	pid	mvec	ntag	nhop
	D	A0	0	0.4	C0	C
∗	D	A1	0	0.5	B0	B

Node B: Forwarding table

	dst	tag	pid	mvec	ntag	nhop
	D	B0	0	0.3	D0	D
∗	D	B1	0	0.2	C0	C

Node C: Forwarding table

dst	tag	pid	mvec	ntag	nhop
D	C0	0	0.1	D0	D

Figure 5: A running example of the compilation algorithm.

Probe sending states. If a physical node X is a valid destination allowed by the policy, then exactly one of its virtual nodes is a *probe sending* state. This state has the form $(X0, \sigma_0(q_{0_0}, X), \cdots, \sigma_k(q_{0_k}, X))$; all probes that originate from X initially carry this state. This is because, when probes start at the originating node, they have only traversed the first hop "X" from the initial automata states q_{0_i}.

Policy compliance. Any path in the PG from an accepting state to a probe sending state is a policy-compliant path. All policy-compliant physical paths also exist in the PG.

4.2 Packet forwarding

Before describing the protocol itself, we first describe the structure of the forwarding (`FwdT`) tables on each switch. The compiler only generates the table layout, and then actual entries are populated at runtime based on link metrics, which we describe in the next subsection.

An entry in the forwarding table has the form `[dst`*`,tag`*`,pid`*`,mv,ntag,nhop]`, where the star fields are table lookup keys. Each row of the table indicates where the given switch will send packets destined for `dst` when those packets carry a PG node `tag` and probe number id `pid`. The sender of packets will set the initial tag and the probe number based on its best path. At each intermediate hop, when a packet with a given `dst`, `tag`, and `pid` matches an entry in `FwdT`, the switch looks up the next tag (`ntag`) and replace the packet's tag with it; it also forwards the packet to the next hop (`nhop`). The metrics vector (`mv`) is not used for packet forwarding, but for populating the entries. A property of `FwdT` is that any `tag`-`ntag` pair in this table corresponds to a PG edge, and when a `ntag` is written into a packet it is then forwarded out the `nhop` port that leads to a topology node corresponding to that `ntag`. This process implies that forwarding will always follow edges in the PG.

As an example, consider the `FwdT` table for switch `B`: the policy allows `B` to reach `D` either through a) `B-D`, satisfying (part of) the regular expression `ABD`, or through b) the best of `B-D`, `B-C-D`, and `B-A-C-D`, satisfying the regular expression `B.*D`. The former corresponds to the virtual node `B0` in the PG, and the latter is implemented by a combination of both `B0` and `B1`. Hence, the reader may observe that it is possible for nodes of the product graph to contribute to the implementation of more than one regular expression in the policy—this sharing improves algorithm performance as a single probe can contribute to uncovering information useful in more than one place in the policy.

Ignoring for now how the forwarding entries were populated, consider the first entry in `B`'s table in Figure 5(e). The entry is generated from the virtual node `B0`: if a packet is at `B` with `tag=B0` and a destination `D`, then either that packet was sent from `A`, and traveled to `B` or it was sent directly from `B`. In either case, the current best path is through the next hop `nhop=D` with a metric `mv=0.3`. Moreover, before `B` sends the packet to `D`, it should update the tag to the new virtual node's tag `D0`. The second entry in `B`'s table is generated from `B1`. When packets are tagged with `B1`, there are two paths they could take to `D`: `B-C-D` and `B-A-C-D`. Currently, the least utilized path is `B-C-D`, so `nhop=C` and `mv=0.2`. The updated tag will then be `C0`. For this policy, only one probe is needed (carrying utilization), so there is only a single probe id (`pid`) of 0. The asterisk next to the `B1` entry indicates that `B` prefers `B-C-D` over `B-D`, which is determined after evaluating the user policy on both paths. Hence, traffic sourced from `B` will choose `B-C-D`. Note that each source can determine its own preference: although `B` prefers `C`, `A` can still use `A-B-D` since `A`'s traffic will be forwarded using the `B0` entry.

Function SwiForwardPkt in Figure 6 summarizes the packet forwarding logic. When a packet first arrives at the switch from a host, it is treated differently. In this case, this first switch must determine the preferred path for the packet (with each path having a representative destination, PG start node and probe id), which is stored in the `BestT` table.

4.3 Sending probes

While the forwarding tables compactly encode how devices should forward traffic in a policy-compliant way, we have yet to describe how these tables are populated. To this end,

```
function InitProbe(PGNode n, ProbeId pid)
    if n.isPrbSendingState then
        p.origin ← ToTopoNode(n)
        p.pid ← pid
        p.tag ← n.tag
        p.mv ← InitMVec
        MulticastProbe(n, p)

function MulticastProbe(PGNode n, Probe p)
    pg_neighbors ← GetPgOutNeighbors(n)
    topo_neighbors← ToTopoNodes(pg_neighbors)
    Multicast(p→topo_neighbors)
```

```
function ProcessProbe(Switch S, Probe p)
    n ← NextPGNode(S, p.tag)
    p.mv ← UpdateMVec(p.inport)
    key ← (p.origin, n.tag, p.pid)
    (mv, ntag, nhop) ← FwdT[key]
    if f(p.pid, p.mv) < f(p.pid, mv) then
        FwdT[key] ← (p.mv, p.tag, p.inport)
        oldKey ← BestT[p.origin]
        if s(key) < s(oldKey) then
            BestT[p.origin] ← key
        p.tag ← n.tag
        MulticastProbe(n, p)
```

```
function SwiForwardPkt(Packet p, Switch S)
    key ← (p.dst, p.tag, p.pid)
    if fromHost(p.inport) then
        key ← BestT[S]
        p.pid ← key.pid
    (mv, ntag, nhop) ← FwdT[key]
    p.tag ← ntag
    SendPkt(p, nhop)
```

Figure 6: Pseudocode for the synthesized switch-local programs. Underlined variables are PG states.

the Contra compiler generates protocol logic for propagating probes from probe sending states in order to populate the tables with the best paths to each destination.

At a high level, each node in the PG propagates probes to its neighbors. For instance, a probe starts at `D0` (`D` with tag 0) and is sent to `B0` and `C0`. `C0` updates the utilization to be 0.1 and adds this entry to its forwarding table before sending a new probe to `A0` and `B1`. Similarly, `B0` adds an entry for the probe it received from `D0` with utilization now 0.3 before sending a new probe to `A1`. `A1` receives a probe from `B0` and adds an entry with utilization 0.5, etc. `A0` receives a probe from `C0` with metric now 0.4 and adds this entry to its table before sending the probe to `C0` and `B1`. Probes will continue to propagate through the PG so long as they decrease the best available metric for that probe type and PG node. Since a static analysis ensures that policy metrics are monotonically increasing, probes will not be propagated endlessly in loops.

To determine which entry to use for forwarding local traffic, switches compute the best path by keeping a pointer to their overall best entry (the asterisks in Figure 5(e)). For example, consider the node `A`. Evaluating the policy on `A0` results in ∞ because `A0` is not an accepting state for regex `ABD` or `B.*D`. On the other hand, evaluating the policy in `A1` results in 0 (the best rank) because `A1` is an accepting state for regex `ABD`. Hence, the asterisk appears by `A1`.

Probe generation. Probes are generated from initial PG states (*e.g.*, `(D0,1,1)` in our example). These sending states use the procedure in InitProbe to initiate probes, and use MulticastProbe to multicast the probes along the outgoing PG edges to all downstream neighbors. Each probe carries four fields: (1) *origin* denotes the topology location of the sending switch (*i.e.*, `D` for the state `(D0,1,1)`); (2) `pid` is the probe id, as obtained from the policy decomposition; (3) `mv` denotes the metrics vector used in the policy (*i.e.*, utilization in the example, which is initialized to a default value 0); and (4) *tag* denotes the id of the PG node the probe is at.

Probe dissemination. The ProcessProbe algorithm describes how a switch processes a probe from its neighbor. It first obtains the PG node for the neighbor (n). Next, it updates the metrics in the probe based on the port at which the probe arrived, *e.g.*, the maximum of the probe's carried utilization and the local port's utilization. If this probe (with id *i* and tag *t*) contains a better metric according to *f* than what is currently associated with *i* and *t*, then it updates its `FwdT` table with the new `nhop`, `ntag`, and `mv` based on this probe. The switch also checks if an update affects its overall best choice (*i.e.*, where the asterisk points to), as recorded in the `BestT`. The switch looks up the existing value and compare it to the current probe using the function *s* that checks the overall value of the probe (not just per tag / probe id). Finally, the probe tag is updated to the correct value for n, and the probe is multicast to all PG neighbors.

5 Compilation: Unstable metrics

Consider using the same solution as described in Section 4, but instead of sending just one probe, sending many probes periodically, one per time interval. This introduces new complications due to the lack of synchronization; certain parts of the network may be working with outdated information. In fact, the example sequence from Section 3, Figure 3(b)-(f) demonstrates exactly how a problem can arise—the example culminates with the forwarding loop S-A-B-S. Notice also that this loop is technically policy-compliant because any path from S to D is allowed, so the packet tagging mechanism would not prohibit it.

The key issue is that when switches use old probes to make decisions, loops can form. In Figure 3(b), the probe *p* from B to A took a long time to propagate; by the time *p* arrived at A, the metrics had already changed again. Concretely, *p* was computed using an old metric u(A-D)=0.1, which had since changed to 0.5; but A still used this outdated probe and thought D was a better next hop.

5.1 Preventing persistent loops

To prevent loops, we draw on ideas from Babel [16], which distinguishes outdated probes from new ones using a version number, and discards outdated probes. In our scenario, this suggests A should discard *p* because it has an older version number, and should continue to use D as the next hop, thereby avoiding the loop. When a round of probes is still in propagation, switches may have temporarily inconsistent views, so a packet may experience a transient (yet policy-compliant) loop. However, versioned probes would guarantee that persistent loops would not form [16].

We note that there is a long body of work on loop prevention in routing protocols with tradeoffs being made in terms of space overhead and convergence time. Contra's compilation algorithm can potentially be integrated with different loop prevention techniques. For example one could prevent loops by adding a bit vector to each probe to record visited nodes (*i.e.*, a path-vector protocol) at the cost of greatly increased probe overhead (one bit for every router). We opt for our approach to limit the space overhead of probes.

Refinement (Versioned probes). *As before, except that a) switches attach version numbers to the probes, which increase for each round; b) the* `FwdT` *table records the version number of the probe that was used to compute each entry; and c) before a switch updates an entry with version* v *with a probe of version* v'*, it needs to check that* $v' \geq v$.

5.2 Probe frequency

Versioning the probes, however, leads to an additional complexity: a node may not always be able to pick the best path. Consider a case where D sends probes to S every 0.2 ms along two available paths: a) p_1 with utilization of 0.4 and a latency of 0.1 ms, and b) p_2 with utilization of 0.1 but a latency of 0.2 ms. Due to the higher latency of p_2, whenever S receives a probe from this path, it would find the probe to be outdated, since newer probes had arrived from p_1. As a result, S ends up always using p_1 which has a higher utilization, even if the policy prefers the least-utilized path p_2.

We observe that this problem can be addressed by ensuring (with high probability) that old probes are fully propagated throughout the network before new probes are sent out. In the above scenario, if we set the probe period to be 0.2 ms or larger, then S would instead pick p_2 to be the better path after both probes have been received.

Refinement (Limited probe frequency). *As before, except that the probe period needs to be larger than or equal to* $0.5 \times RTT$*, where* RTT *is the highest round-trip time between any pair of switches in the network.*

5.3 Policy-aware flowlet switching

Since Contra can spread traffic in the same flow across multiple paths, it is important to mitigate the potential out-of-order packet delivery. One classic approach is *flowlet switching* [44], where packets in the same flow are grouped in bursts/flowlets and the same forwarding decision is applied to the entire flowlet. By doing so, the first packet in the flowlet is always forwarded to the best path, and subsequent packets in the same flowlet would inherit this (slightly outdated) forwarding decision. This also increases network stability: although each switch's best path is constantly fluctuating, at any given point, much of the current network traffic is pinned to a particular path. Only new flowlets will make use of the current path information.

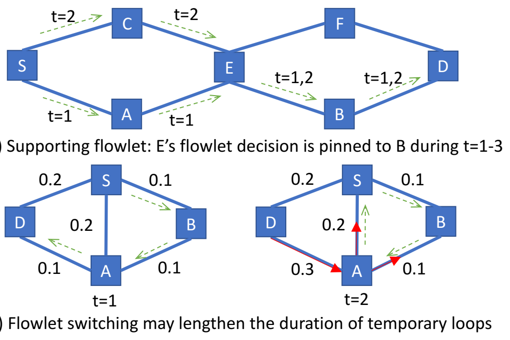

t=2
C
t=2
F
S
E
t=1,2
t=1,2
D
t=1
A
t=1
B
(a) Supporting flowlet: E's flowlet decision is pinned to B during t=1-3
0.2
S
0.1
D
0.2
B
0.1
A
0.1
t=1
0.2
S
0.1
D
0.2
B
0.3
A
0.1
t=2
(b) Flowlet switching may lengthen the duration of temporary loops

Figure 7: Challenges due to flowlet switching.

A first attempt to implement flowlet switching in Contra would be to have each switch maintains a table of the form `[fid*,nhop,t]`, where `fid` is the flowlet ID (from hashing a packet's five tuple), `nhop` is the temporarily "pinned" next hop, and `t` is the timestamp of the last packet in `fid`. When the next packet in `fid` arrives, the switch computes the gap between its timestamp and `t`: if the gap is small, this packet will use the current `nhop`; otherwise, the switch expires this entry and starts a new flowlet. Perhaps surprisingly, deploying such a flowlet switching mechanism with Contra may result in policy violations. Consider the example in Figure 7(a), where the policy prefers the least utilized of the upper or lower paths, but avoids the "zigzag" path.

```
if SCEFD + SAEBD then path.util else ∞
```

Suppose that at t=1, S sends traffic to D via the lower path due to its lower utilization; using flowlet switching, all switches temporarily pin this flowlet to their respective next hops along the path when they receive the first packet in the flowlet (*e.g.*, A pins to E at t=1.1, which expires at t=2.1; E pins to B at t=1.2, which expires at t=2.2; and so forth). At t=2, S discovers that the utilization of the upper path has improved, and changes its preference to D instead. However, if the packets from S arrive at E before t=2.2, which is its flowlet switching expiration time, E will continue to forward these packets to the lower path, causing a policy violation.

The fundamental reason for this is that flowlet switching is oblivious to routing constraints. Our solution makes it *policy-aware* by adding PG tags to flowlet entries. Concretely, policy-aware flowlet switching extends the table format to be `[tag*,pid*,fid*,nhop,t]`, where `tag` and `pid` are obtained from the probe that created the forwarding entry, and `tag`, `pid`, and `fid` are match keys. This enables flowlet switching *within each policy constraint and probe type*. Now, when E processes the packet at t=2.2, it would see that the packet was constrained to traverse the upper path and use the flowlet entry for that path.

Refinement (Policy-aware flowlet switching). *As before, except that switches perform policy-aware flowlet switching by maintaining multiple entries for the same flowlet, each for a different path constraint/tag and probe type.*

5.4 Handling failures

Switches also need to discover new best paths when links or switches fail. Suppose that the best path for S to reach D is S-A-D, but the link A-D goes down at some point. We need to ensure that S will learn about the failure and change to another available path if one exists. Our solution is to first detect failed links, and then to expire flowlet entries when their next hop is along a link that is believed to be failed.

Refinement (Expiration). *As before, except that a flowlet entry is expired when a packet arrives at a switch and is going to be forwarded by the flowlet entry, and the next hop is along a failed link.*

Handling failures, of course, requires the existence of a failure detection mechanism. The specific link failure detection methods are beyond the scope of Contra; the above approach merely ensures that switch routes around detected failures for future flowlets. In our implementation of Contra, a switch marks a link as failed when there have been no probes along the link for *k* probe periods, where *k* is a parameter that determines how fast failures should be discovered.

5.5 Breaking transient loops

As we discussed, transient loops may still occur when probes are in propagation. Figure 7(b) is a concrete example. At t=1, the best path for S to reach D is S-B-A-D. Then, at t=2, A receives a probe from D carrying a worse metric, so it propagates the probe to S and B. Before this probe arrives at S and B, A learns of the better path through S, and traffic that is already in flight will be forwarded along a transient loop S-B-A-S; this loop will be broken once S and B receive the new probe because it has a higher version number.

Interestingly, flowlet switching may lengthen the duration of transient loops because flowlet switching decisions may expire at different times across hops. Suppose that A's timer expires at t=3, and it starts using the new best next hop S to reach D; however, the timers at S and B do not expire until t=4. Then the traffic would continue to be forwarded in the loop S-B-A-S regardless of the newer probe, until S and B have updated their flowlet switching decisions.

We address this by detecting loops lazily and flushing the offending flowlet switching entries upon detection. Concretely, each switch maintains a *loop detection* table {`flow_hash`*,`maxttl`,`minttl`}, which maps a flow's CRC hash to the maximum and minimum TTL values seen at this switch. δ=`maxttl-minttl` should be stable in the absence of loops: it is the difference between the longest and the shortest paths packets could have traversed to reach the current switch. However, when there is a loop, δ would continue to grow. Therefore, switch detects a potential loop (with false positives) when its δ exceeds a threshold. When this happens, the switch expires its flowlet switching decision, and starts a new flowlet using the latest metric in the `FwdT` table. Hence, we arrive at our final solution below.

Final solution. *As before, except that switches use loop detection tables to detect and break loops by refreshing their flowlet switching decisions using the latest metrics.*

6 Evaluation

We aim to answer three main questions in our evaluation: a) How well does Contra scale to large networks? b) How competitive is Contra compared to hand-crafted systems? and c) How well does Contra work on general topologies? Due to space constraints, some results appear in the Appendix.

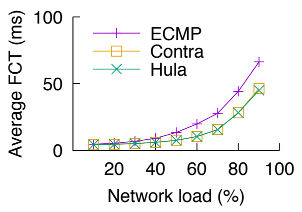
6.1 Prototype implementation and setup

Our prototype [3] consists of 7485 lines of code in F# [6], which processes policy and topology descriptions, and generates switch-local P4 programs. The compilation also minimizes the number of tags and forwarding table sizes.

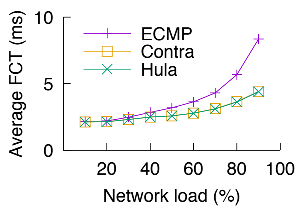
Experimental setup. We have used three types of topologies: a) data center topologies, b) random graphs, and c) real-world topologies (e.g., Abilene [1] and those from Topology Zoo [7]). Our baseline systems for data center networks are ECMP and Hula [30], load-balancing schemes for a Fattree topology. ECMP balances traffic randomly without considering network load, and Hula is load-aware and always chooses the least-utilized path among all shortest paths. Our baseline system for arbitrary graphs is SPAIN [35], which statically (*i.e.*, independently of network load) selects multiple paths to route traffic. We used two workloads obtained from production networks for our evaluation: a web search workload [12], and a cache workload [43].

Simulation vs. Emulation. For simulation, we have used a customized version of ns-3 [4] that implements P4 switches using the `bmv2` model, and it runs on a Dell server with six Intel i7-8700 CPU cores and 16 GB of RAM. For emulation, we have used the `bmv2` switches in a Mininet [34] cluster on 15 CloudLab [17] servers, each with eight cores and 128 GB of RAM. The simulation allows us to conduct experiments at high link speeds (up to 40 Gbps), as traffic forwarding is simulated in an idealized environment. The emulation, on the other hand, generates and forwards real network traffic across machines. This allows us to evaluate the systems in a high-fidelity environment [34], albeit with lower link speeds (50 Mbps) to avoid causing bottlenecks in the software switches.

We have replicated the emulation setup (45 switches in a 6-ary Fattree) in our simulator, and measured for eight setups (workloads+policies) the Pearson correlation coefficient (PCC) [42] between the emulated and simulated results. Seven of the setups produced PCC values of 0.99+, and the other produced 0.98 (1.0 means perfect correlation). This "meta" experiment confirms that the simulation and emulation results closely mirror each other, and that our observations are consistent across setups. Below, after presenting compiler results (§6.2), we first describe our simulation results (§6.3-§6.5), and then the high-fidelity emulation (§6.6).

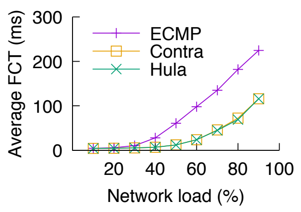


(a) The web search workload

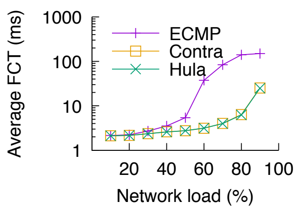


(b) The cache workload

Figure 8: Contra achieves a similar FCT as Hula, outperforming ECMP considerably.



(a) The web search workload



(b) The cache workload

Figure 9: Contra achieves a significantly shorter FCT than ECMP on an asymmetric topology with a failed link.

6.2 Compiler scalability

To test the scalability of our compiler, we used topologies of varying sizes from 20 to 500 nodes. For each topology, we evaluated three different policies: a) minimum utilization (MU: no regular expressions, single performance metric), b) waypointing (WP: three regular expressions, single performance metric), and c) congestion-aware routing (CA: no regular expression, non-isotonic policy with two performance metrics). The concrete numbers are included in Figure 14 in Appendix C, and we summarize the key takeaways here: The compiler scales roughly linearly with topology size, and completes in seconds on topologies with hundreds of nodes. Use of regular expressions increases product graph size and hence compilation time. Non-isotonic policies add some overhead due to the additional policy analysis.

We have also measured the switch state used by the generated P4 programs (Figure 15 in Appendix C). At a high level, WP and CA require more state than MU: WP's regular expressions require tracking automaton states. and CA's non-isotonic policy requires a separate table for each metric in the decomposed policy (*i.e.*, separate entries for different `pid` values). However, no more than 70 kB of switch state was necessary in any experiment—a tiny fraction of the available memory on modern switches (tens of megabytes) [2].

6.3 Performance: Data center topology

We compare Contra with ECMP and Hula in terms of their flow completion time (FCT) in simulation. In our topology (Figure 13a), we used 32 hosts with 10 Gbps links, with 10 Gbps links between switches, and an oversubscription ratio of 4:1. Half of these hosts were configured as senders, and the other half receivers. We set the probe period to 256μs and flowlet timeout to 200μs for both Contra and Hula. All links have a queue buffer size of 1000 MSS by default. Moreover, we tuned the desired network load from 10% to 90% by adjusting the flow arrival times, and obtained the FCT for each setting. The policy used in Contra is widest shortest paths (WSP; policy P3 in Table 1), which picks the least-utilized shortest paths and is equivalent to Hula; we found that the performance of Contra with the MU policy (P2 in Table 1) is similar to WSP in this setup.

Symmetric Fattrees. Figure 8 shows that both Contra and Hula outperform ECMP considerably because they balance traffic based on network load. At 90% load, they reduce the average FCT by 30% for the web search dataset and by 47% for the cache dataset. Hula outperforms Contra slightly, by 0.33% on average across different datasets and network loads. This is because Hula knows statically what paths are shortest paths (and hence what ports to send probes from), whereas Contra has to discover this information dynamically (*i.e.*, by carrying the path length as well as the utilization, and also by sending probes both "up" and "down" at each level in the datacenter)—hence Contra sends more probes than Hula in order to achieve generality over different topologies and policies. Further compiler optimizations could likely reduce this gap further (*e.g.*, by identifying shortest paths statically). We also refer interested readers to Appendix I for detailed breakdown of probe and tag overheads.

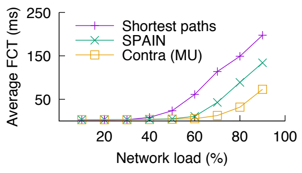
Asymmetric Fattrees. Next, we ran the same experiment after injecting a failure on a link between an aggregation switch and a core switch, so that the topology became asymmetric. Figure 9 shows the FCT for this setting. In this case, we found that ECMP incurred heavy traffic loss beyond 50% network load, even though 75% of all capacity remains after the link failure. The average FCT increased by 3.18$\times$ for the web search dataset and 8.72$\times$ for the cache dataset. In contrast, Contra and Hula only had an increase of 1.80$\times$ for the web search dataset and 1.67$\times$ for the cache dataset, relative to the FCTs on the symmetric topology.

We further measured the queue growths under ECMP and Contra with 60% workload on the web search dataset without bounding the maximum queue sizes (see Figure 16 in Appendix D). We found that Contra's queue lengths never exceeded 1000 MSS, whereas ECMP saw queue lengths larger than 1000 MSS more than 97% of the time, which can cause heavy traffic loss when the queues are full.

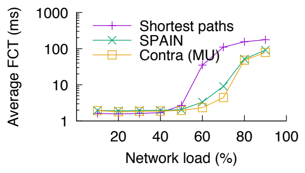
In order to measure Contra's response time to link failures, we sent UDP workloads at 4.25 Gbps rate, and brought down an aggregate-core link. Contra successfully detected the link failure after 800μs, which is close to the failure detection threshold (3$\times$probe period=768μs) that we used. Upon detection, Contra routed around the failure and recovered the throughput within 1 ms. We found that Hula performs similarly as Contra (Figure 17 in Appendix D).

6.4 Performance: Arbitrary topologies

We now evaluate the performance of Contra on general topologies. We modeled our network after the Abilene [1] topology, configured all links to be 40 Gbps, and randomly chose four pairs of senders/receivers. Since Hula is specialized to a Fattree topology and will not work outside of this context, and since ECMP will not load balance when there is only a single shortest path, we have used two other baselines: a) shortest path routing (SP), which simply sends traffic to the shortest paths, and b) SPAIN [35], which precomputes all paths using (static) heuristics that avoid overlap, and then load balances between these paths.



(a) The web search workload



(b) The cache workload

Figure 10: Contra outperforms SPAIN in FCT.

Figure 10 shows the FCT for these different systems. A naïve strategy that simply chooses shortest paths performs the worst. Since SPAIN can utilize multipath routing, it outperforms SP by 32.5% on average for the web search workload and 26.9% on for the cache workload. Contra achieves the best performance among the three: it evenly distributes traffic based on path utilization, and reduces FCT relative to SPAIN by 31.3% on average for the web search workload and 13.8% for the cache workload.

6.5 Protocol dynamics

Next, we study the network dynamics of performance-aware routing. The high-level note here is that, if a policy uses `m` as the metric, then paths with better `m` values may not necessarily be shorter and have lower end-to-end latency. The MU policy is a case in point, because 1) a least-utilized path could be a non-shortest path; and 2) compared to a slightly more utilized shortest path, the non-shortest path may also have higher end-to-end delay if its extra propagation delay offsets its lower queueing delay. Finally, when nodes are temporarily out of sync (§5.5), transient loops may arise.

Transient loops. We first quantify the amount of packets forwarded in transient loops. Under the MU policy on the Fattree and Abilene topologies at 60% load, only tiny fractions of traffic (0.024% and 0.021%, respectively) experienced loops. Compared to the shortest paths, these packets traversed 3.15 more hops on average for the Fattree, and 3.09 for Abilene. They also experienced an increase in end-to-end latency by 72.4μs (Fattree) and 65.2μs (Abilene). Our loop detection mechanisms successfully broke these loops.

Non-shortest paths. Packets could also traverse non-shortest paths without experiencing loops. We observed that the fractions of such packets are 16.6% for Fattree and 39.8% for Abilene. These packets also experienced latency increases of 42.3μs (Fattree) and 39.3μs (Abilene). An interesting observation here is that the application performance (in terms of FCT) depends on the tradeoff between propagation delay and queueing delay. In other words, if least-utilized paths happen to have high propagation delays, then the MU policy may not always lead to FCT improvement. We have observed both cases in our experiments with different topologies, path latencies, and number of shortest paths.

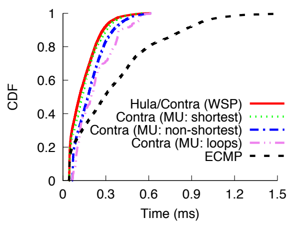
Contra, Hula, ECMP. Figure 11a shows the end-to-end latency packets experienced in Contra for two policies (WSP and MU) on the Fattree, and shows the breakdown for the MU policy (shortest paths, non-shortest paths, and transient loops). As expected, packets never experienced loops under WSP, which routes traffic to the least utilized of shortest paths. For Contra (MU), packets forwarded in loops took more time than those on non-shortest paths, and both took longer than those on shortest paths; they experienced slightly higher latency than in Hula. Nevertheless, Contra (MU) still outperforms ECMP significantly, because shortest paths with high utilization in ECMP have higher queuing delays than non-shortest paths with low utilization. (Results for Abilene are similar; see Figure 19a in Appendix E.)

Network loads. We also found that more packets tend to traverse longer paths as the network load increases (Figures 19b and 19c; in Appendix E). Across all scenarios, 12%-16% traffic took longer paths in the Fattree; within such traffic, 93%-98% took 2 extra hops and the rest took 4 extra hops. On Abilene, where shortest paths are fewer, 32%-50% traffic took longer paths; within this, 97%-99% took fewer than 4 extra hops and the longest path had 9 extra hops.

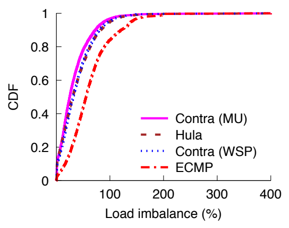
Load imbalance. Next, we focus on understanding the load imbalance over small timescales in ECMP, Hula, and Contra. Figure 11b shows the CDF of load imbalance of the four aggregate-core links on the Fattree topology at 70% load. We measured the throughput of these links for each 100μs interval for 1 second; we then computed the throughput differences between the most and least loaded links, and normalized them by the average throughput across the links. As we can see, Contra and Hula perform similarly, and they both balance the load much more evenly than ECMP. In particular, this shows that packets traversing non-shortest paths or transient loops in Contra do not lead to notable load imbalance even at small timescales.

6.6 High-fidelity emulation

We have set up a high-fidelity emulator on 15 CloudLab [17] servers using a distributed cluster of Mininet [34]. Our topology is a 6-ary Fattree with 45 switches running P4 `bmv2` [5] and 57 end-hosts with 1:1 oversubscription. We have set the link speeds to be 50 Mbps and verified that this is the highest link speed achievable in our testbed without causing bottle-



(a) End-to-end latency



(b) Load imbalance

Figure 11: Packets that experience transient loops or non-shortest paths spent more time in the network (a); but they do not lead to notable load imbalance (b).

necks in the software switches. As discussed before, we have mirrored the same setup in simulation and conducted experiments in both settings. The highest-level takeway is that we observed similar results on both setups. Below, we summarize the key findings, and note that the full set of figures (Figures 20-25) are in Appendix F.

Symmetric Fattree. Contra and Hula outperform ECMP considerably in FCT, and they perform similar to each other. At 90% load for the web search workload, they improve the performance of small flows (<100 kB) by 21%, large flows (>10 MB) by 12.7%, and 13.7% across flows.

Asymmetric Fattree. We then re-ran the FCT experiments, after bringing down three out of 27 aggregate-core links (11.1% reduction to overall capacity). Compared to the topology without failures, at 90% load, the average FCT increased by 15% for the web search dataset and 23% for the cache dataset in ECMP. Contra, on the other hand, only increased by 6% (web search) and 9% (cache), respectively.

Arbitrary topologies. We have also set up the Abilene topology in emulation and compared Contra with SPAIN, and note again that Hula only works on a tree topology. We found that at 90% load, Contra reduces FCT by 22.0% (web search) and 45.7% (cache) under the MU policy. For the web search (cache) dataset, it achieves 11.0% (36.5%) speedup for small flows, and 20.9% (46.1%) speedup for large flows.

7 Related Work

Traffic engineering and load balancing. Contra is different from centralized TE solutions, such as B4 [26], SWAN [25], Hedera [9], MicroTE [15], and Gvozdiev [23], as well as distributed TE solutions, such as TeXCP [28], MATE [18], and Halo [33], in that Contra performs fine-grained load balancing in the data plane. Contra borrows a similar load balancing mechanism from Hula [30] and Conga [11], so their characteristics are similar in terms of dynamics and performance benefits. The main novelty of Contra over all of these systems is to generalize point solutions to a wide range of policies and arbitrary topologies.

Routing. Existing work has studied loop prevention in distance-vector routing [21, 36, 10, 39, 16, 38] with different overhead, convergence, and stability tradeoffs. Contra is most related to DSDV [39], AODV [38], and Babel [16], which use sequence numbers on route updates for convergence. The novelty of Contra lies in its use of programmable data planes to implement a wide array of distance-vector protocols in the presence of unstable metrics, and its design of policy-aware flowlet switching mechanisms.

Regular languages for networking. NetKAT [13], Merlin [45], FatTire [40], and Propane [14] all use regular expressions to specify path constraints, but none of them supports dynamic preferences based on network conditions.

8 Discussion

Correctness guarantees. Contra guarantees that traffic always follows policy-compliant paths even in the presence of unstable metrics. In terms of performance, previous work has shown that when switches make distributed decisions, the resulting mechanism is not globally optimal [11]. However, as demonstrated in our experiments as well as previous work [11, 30], performance-aware routing still leads to FCT improvements compared to static routing.

Policy changes. Policy changes can be handled by recompilation, which would generate new switch programs. As we demonstrated in the experiments, policy compilation is fast and scales to large networks. We expect policy changes to happen infrequently and only on a larger timescale. In order to implement the updates, Contra may be able to borrow existing work on consistent update algorithms [41].

Traffic classes. Contra currently does not support traffic classes. Adding such support would require extending the language with header predicates [20, 13], and designing new mechanisms to prioritize one traffic class over another.

9 Conclusion

We have presented Contra, a system for specifying and enforcing performance-aware routing policies. Policies in Contra are written in a declarative language, and compiled to switch programs that run on the data plane to implement a variant of distance-vector protocols. These programs generate probes to collect path metrics, and dynamically choose the best paths along which to forward traffic. Our evaluation shows that Contra scales well to large topologies, and that the synthesized switch programs can achieve performance competitive with hand-crafted solutions that are specialized to particular topologies and hard-coded policies. Contra is also substantially more general, supporting a wide range of policies over arbitrary topologies.

Acknowledgments: We thank our shepherd Aurojit Panda, the anonymous reviewers, Mina Arashloo, Wei Bai, Shir Landau Feibish, Arpit Gupta, Rob Harrison, Robert MacDavid, and João Sobrinho for their helpful comments and suggestions. This work was supported in part by NSF grants CNS-1703493, FMitF-1837030, and CNS-1801884.

References

[1] Abilene network. `https://web.archive.org/web/20120324103518/http://www.internet2.edu/pubs/200502-IS-AN.pdf`.

[2] Barefoot Tofino Switch. `https://www.barefootnetworks.com/technology/#tofino`.

[3] Contra prototype repository. `https://github.com/alex1230608/contra`.

[4] NS-3 simulator. `https://www.nsnam.org/`.

[5] P4 behavioral model. `https://github.com/p4lang/behavioral-model`.

[6] The F# Functional Programming Language. `http://fsharp.org/`.

[7] The Internet Topology Zoo. `http://www.topology-zoo.org/`.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. SIGCOMM*, 2008.

[9] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. NSDI*, 2010.

[10] B. Albrightson, J. Garcia-Luna-Aceves, and J. Boyle. EIGRP – a fast routing protocol based on distance vectors. In *Proc. Networld/Interop*, 1994.

[11] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proc. SIGCOMM*, 2014.

[12] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. SIGCOMM*, 2010.

[13] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[14] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proc. SIGCOMM*, 2016.

[15] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *Proc. CoNEXT*, 2011.

[16] J. Chroboczek. The Babel routing protocol. RFC 6126.

[17] CloudLab. `https://www.cloudlab.us/`.

[18] A. Elwalid, C. Jin, S. Low, and I. Widjaja. MATE: MPLS adaptive traffic engineering. In *Proc. INFOCOM*, 2001.

[19] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proc. INFOCOM*, 2000.

[20] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proc. ICFP*, 2011.

[21] J. J. Garcia-Lunes-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking*, 1(1), 1993.

[22] T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proc. SIGCOMM*, 2005.

[23] N. Gvozdiev, S. Vissicchio, B. Karp, and M. Handley. On low-latency-capable topologies, and their impact on the design of intra-domain routing. In *Proc. SIGCOMM*, 2018.

[24] C. Hedrick. Routing Information Protocol. RFC 1058.

[25] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *Proc. SIGCOMM*, 2013.

[26] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proc. SIGCOMM*, 2013.

[27] U. Javed, M. Suchara, J. He, and J. Rexford. Multipath protocol for delay-sensitive traffic. In *Proc. COMSNETS*, 2009.

[28] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *Proc. SIGCOMM*, 2005.

[29] G. Karakostas. Faster approximation schemes for fractional multicommodity flow problems. *ACM Trans. Algorithms*, 4(1):13:1–13:17, Mar. 2008.

[30] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable load balancing using programmable data planes. In *Proc. SOSR*, 2016.

[31] P. Kumar, C. Yu, Y. Yuan, N. Foster, R. Kleinberg, and R. Soulé. YATES: Rapid prototyping for traffic engineering systems. In *Proc. SOSR*, 2018.

[32] Q. Ma and P. Steenkiste. Quality-of-service routing for traffic with performance guarantees. In *Proc. IFIP Workshop on Quality of Service*, 1997.

[33] N. Michael and A. Tang. Halo: Hop-by-hop adaptive link-state optimal routing. *IEEE/ACM Transactions on Networking*, 23(6), 2014.

[34] Mininet. `http://mininet.org/`.

[35] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center Ethernet for multipathing over arbitrary topologies. In *Proc. NSDI*, 2010.

[36] S. Murthy and J.J.Garcia-Luna-Aceves. A loop-free routing protocol for large-scale internets using distance vectors. *Computer Communications*, 21(2), 1998.

[37] D. Pei, X. Zhao, D. Massey, and L. Zhang. A study of BGP path vector route looping behavior. In *Proc. ICDCS*, 2004.

[38] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561.

[39] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proc. SIGCOMM*, 1994.

[40] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative fault tolerance for software-defined networks. In *Proc. HotSDN*, 2013.

[41] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

[42] J. L. Rodgers and W. A. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, 1988.

[43] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proc. SIGCOMM*, 2015.

[44] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCP's burstiness with flowlet switching. In *Proc. HotNets*, 2004.

[45] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *Proc. CoNEXT*, 2014.

[46] Y. Wang and Z. Wang. Explicit routing algorithms for internet traffic engineering. In *Proc. ICCCN*, 1999.

[47] M. Zhang, B. Liu, and B. Zhang. Multi-commodity flow traffic engineering with hybrid MPLS/OSPF routing. In *Proc. GLOBECOM*, 2009.

A Policy analysis and decomposition

At a high level, the Contra compiler implements the policies in a distance-vector protocol, where switches propagate periodic probes and compute a best next hop for each destination using the path metrics. To avoid flooding the network with probes, a switch will only disseminate the best probe in a batch and discard the rest. Moreover, if a policy uses multiple metrics, each probe will carry all metrics to further reduce traffic. However, these techniques are not always safe—the policy needs to be *isotonic*, because otherwise downstream switches can wind up with suboptimal paths. The policy also needs to be *monotonic*, because otherwise loops may form.

Monotonicity. A policy f is monotonic iff. extending a path p by an additional link l does not result in a better ranked path, i.e., $f(p) \le f(p \cdot l)$; f is *strictly* monotonic if $f(p) < f(p \cdot l)$. Strict monotonicity ensures that loops will not form in distance-vector protocols (assuming static metrics that do not change), because a path's rank only degrades as it gets longer [22]. In principle, one could write a policy that is not monotonic, such as `minimize (- path.len)`, but in practice, we are not aware of such policies actually in use. On the other hand, there are practical policies such as `minimize (path.util)` that are not *strictly* monotonic. To ensure safety, the Contra compiler implements a conservative monotonicity analysis and alerts a programmer of a potential error if the policy is non-monotonic. But our compiler accepts non-strict monotonic programs: our probe propagation mechanism associates an "age" with each probe stored in a switch, and break ties by rejecting more recent probes if they have the same value as the currently used metric, because they may have traversed zero-weight cycles.

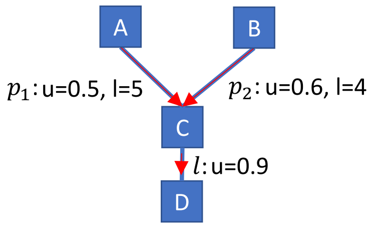


Figure 12: Contra requires (sub)policies to be isotonic.

Isotonicity. A policy f is isotonic iff. for any paths p_1, p_2, and any link l, extending both paths by l preserves the original relative ranking, *i.e.*, $f(p_1) \le f(p_2) \iff f(p_1 \cdot l) \le f(p_2 \cdot l)$. Isotonicity guarantees convergence to the best paths [22] even if a switch discards suboptimal probes. Figure 12 demonstrates the idea: if C prefers the probe from path p_1 over that from p_2 and discards the latter, then its downstream neighbor D must have the same preference, or else it would miss a path with a better metric. However, there are some useful policies that are non-isotonic, such as the following congestion-aware routing policy [27] that switches between metrics depending on the network condition.

$$\texttt{if } \texttt{path.util} < .8 \texttt{ then } (1, \texttt{path.util}) \texttt{ else } (2, \texttt{path.len})$$

To see why the policy is non-isotonic, consider the switch C in Figure 12 that receives two probes with metrics {`u=0.5,l=5`} and {`u=0.6,l=4`}. C prefers the first probe because `path.util < 0.8` evaluates to true for both probes and the two probes will be ranked based on utilization. However, C cannot simply discard the second probe, because all paths to its downstream neighbor D may be highly congested (e.g., u(D-S)=0.9). In this case, `path.util < 0.8` evaluates to false at D for both probes, causing D's preference to be inverted.

Policy decomposition. The Contra compiler tries to decompose non-isotonic policies into multiple isotonic (and monotonic) subpolicies, and generates different types of probes to propagate each subpolicy. If such a decomposition is impossible, then it rejects the policy. For instance, the compiler decomposes the previous policy as follows:

$$\texttt{if } \boxed{\texttt{path.util}}^{0} < .8 \texttt{ then } (1, \boxed{\texttt{path.util}}^{0}) \texttt{ else } (2, \boxed{\texttt{path.len}}^{1})$$

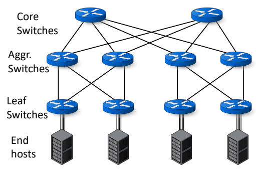
where type-0 probes carry `path.util`, and type-1 probes carry `path.len`. Switches can discard suboptimal probes within each type, but must propagate both types of probes. The complete policy is only evaluated at source nodes. We only attempt this analysis on conditional policies, such as the one above. There remain non-isotonic policies such as "shortest widest paths" (path.util, path.len) that the Contra compiler is unable to implement.

More generally, our compiler performs an analysis to try to decompose f to a collection of subpolicies $(s, f_1, \ldots, f_n)$, where each f_i is monotonic and isotonic, and s combines the subpolicies such that $f(p) = s(f_1(p), \ldots, f_n(p))$. For this decomposition to be correct, s needs to be strictly increasing in each of its arguments, i.e., for any $x_i \le x'_i$, we need to have $s(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, x_n) \le s(x_1, \ldots, x_{i-1}, x'_i, x_{i+1}, x_n)$. Intuitively, this condition allows a switch to safely discard any non-minimum x_i values of each probe type.

Limitations. Currently, Contra does not support traffic classification, but extending the language with header predicates as in prior work [20, 13] should not present any significant intellectual challenge. A more notable limitation involves policies that prioritize one traffic class over another. For instance, B4 [26] prioritizes small, latency-sensitive user requests over large, latency-insensitive bulk transfers. Currently, Contra ranks paths and selects the best path for each flowlet, but does not compare different types of traffic in order to prefer one over the other. We leave integration of such policies into our framework to future work.

B Key topologies for experimental evaluation

We briefly talked about our experimental setups in Section 6.1. Here, we provide more details of topologies we



(a) The Fattree topology

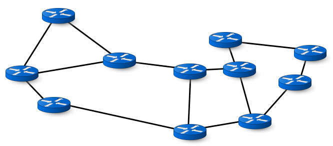
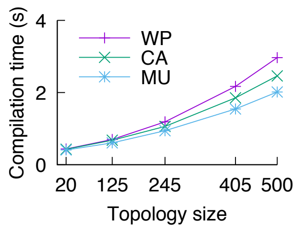

(b) The Abilene topology

Figure 13: The topologies we have used in the simulation environment.

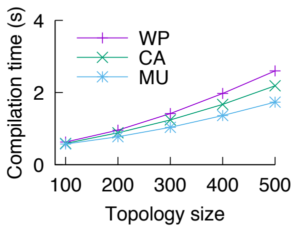
have used. We conducted our experiments in simulation (ns-3) and emulation (bmv2 in Mininet) using two topologies—a Fattree network and the Abilene network. Figure 13 shows topologies we used in simulation setup. The links between switches in both topologies operate at high speed: 10 Gbps in Fattree and 40 Gbps in Abilene . The emulation environment, on the other hand, uses a 6-ary Fattree with 45 switches (not shown, see [8] for more details), as well as the Abilene topology (Figure 13b). In both emulated topologies, the links operate at lower speed (50 Mbps) to avoid causing bottlenecks in the software switches.

C Compiler scalability

We tested Contra compiler scalability (more details in Section 6.2) using topologies of varying sizes from 20 to 500 nodes. Figure 14 shows the time to compile P4 programs from high-level policies as topology size increases. Figure 15 shows the amount of resources used by the P4 programs compiled for different policies.



(a) Fattree topologies

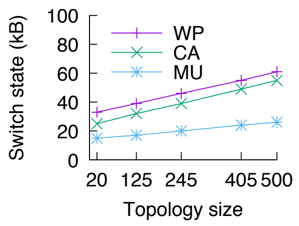


(b) Random networks

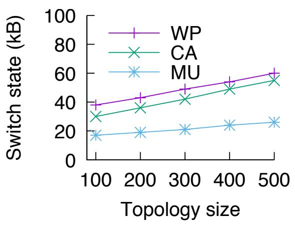
Figure 14: The Contra compiler scales well to large network sizes and sophisticated policies (unit: seconds).

D Link failure

To understand the degree of congestion when topology becomes asymmetric due to link failures, we injected a link failure between aggregation and core switches, and then measured queue lengths of other links. Figure 16 shows the CDF of queue lengths for Contra (under the WSP policy: widest shortest paths) and ECMP when there is a link failure. As we can see, Contra has much shorter queues than ECMP, and Contra queue length is less than 1000 MSS. Note that we have not bounded to queue sizes in order to study the queue growths in different systems. Figure 17 shows the aggregate throughput before and after a link failure. Contra successfully detected this failure in 800 μs and recovered its throughput in 1 ms.



(a) Fattree topologies



(b) Random networks

Figure 15: The Contra compiler generates programs with low memory overhead (unit: kB).

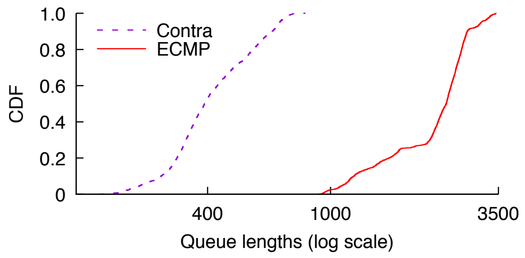


Figure 16: Contra has shorter queues than ECMP.

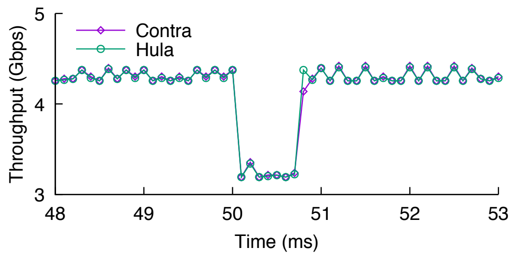
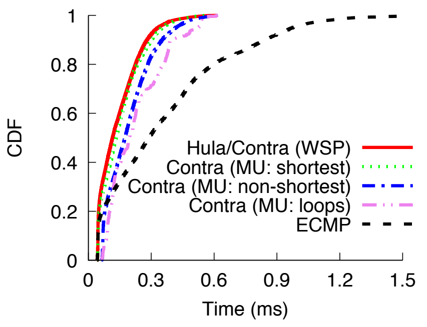


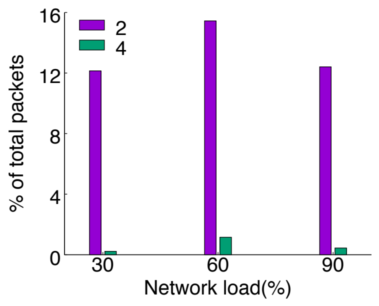
Figure 17: Contra recovers from the link failure within 1 ms.

E Protocol dynamics

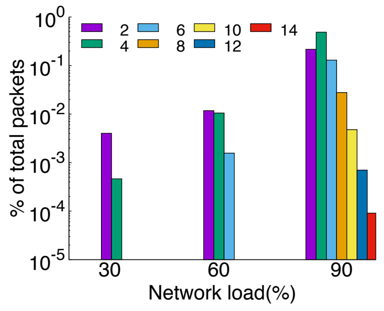
Previously, in Section 6.5 we have summarized the results of network dynamics experiments. Here, we present the full results across network loads. Figures 18a and 19a show the end-to-end latency packets experience in the Fattree topology and the Abilene topology at 60% network load (dataset: web search). For MU policy, we further break down the number of extra hops in transient loops and non-shortest paths for both topologies: Figures 18b and 19b show number of extra hops in non-shortest paths as network load increases; Figures 18c and 19c show the number of extra hops in transient loops.



(a) End-to-end latency.

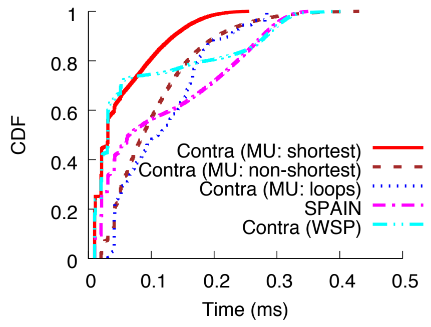


(b) #Extra hops in non-shortest paths

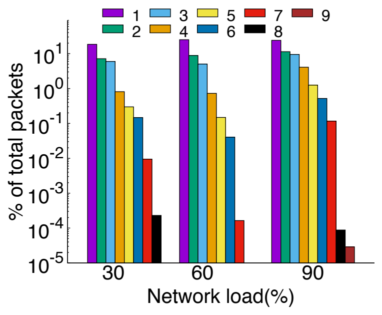


(c) #Extra hops in transient loops

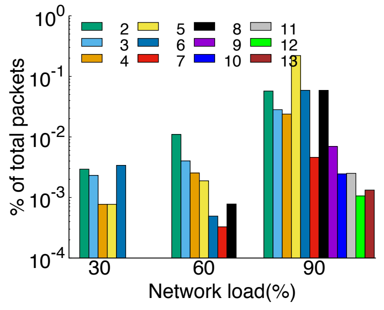
Figure 18: (a): Contra (MU) packets that traverse transient loops and non-shortest paths spent more time in the Fattree network when traffic load is 60%. (b)-(c): breakdowns of extra hops as network load increases; the numbers in the legends denote numbers of extra hops.



(a) End-to-end latency



(b) #Extra hops in non-shortest paths



(c) #Extra hops in transient loops

Figure 19: (a): Contra (MU) packets that traverse transient loops and non-shortest paths spent more time in the Abilene network when traffic load is 60%. (b)-(c): breakdowns of extra hops as network load increases; the numbers in the legends denote numbers of extra hops.

F High-fidelity emulation

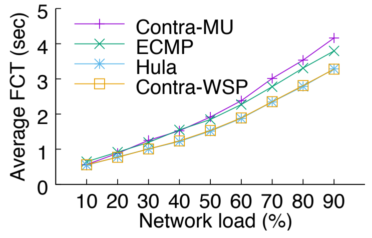
In the main paper, Section 6.6 already summarized the key FCT results obtained in our emulation testbed. Here, we show the full results across workloads, network loads, and Contra policies. All results are obtained over four runs. Figure 20 and Figure 22 show the FCT results for web search and cache workloads in 6-ary symmetric Fattree topology. Figure 21 and Figure 23 show the FCT results for web search and cache workloads in 6-ary asymmetric Fattree topology. Figure 24 and Figure 25 show the FCT results for web search and cache workloads in the Abilene topology.

G Comparison with the FMCF solution

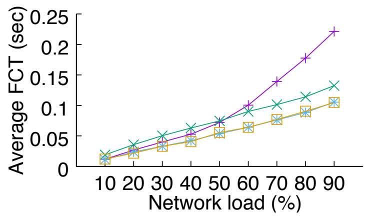
Although the experiments on flow completion times already demonstrate that Contra can boost application performance, we would like to further investigate how Contra performs when compared to an *idealized* solution for which we can derive an optimal bound. To this end, we use a *Fractional Multi-Commodity Flow* problem (FMCF) [29] to model this scenario, and note that similar formulations have been used in other projects [31, 47]. An MCF problem takes as input the (fixed) demand for sender/receiver pairs and the network topology, and computes the optimal traffic splitting across paths in order to minimize the utilization of the most congested link. The *fractional* version of MCF simply means that a flow can be split across different paths as well. This formulation makes several simplifying assumptions, which require minor modifications to the tested systems. Nevertheless, we believe that the results we obtain are still illustrative, as these assumptions make it possible to derive an optimal solution to compare against. The policy we have used in Contra is WSP (widest shortest paths) for Fattree, and MU (Minimum Utilization) for Abilene.

G.1 The FMCF formulation

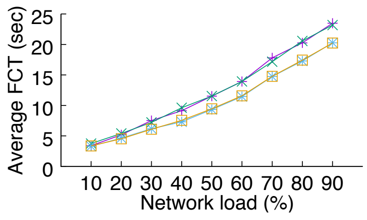
We have used the same formulation as the Linear Programming Formulation (LPF) in [46]. This formulation models the physical network as $G(V,E)$, where V denotes the set of switches and E denotes the set of links. For each link (i,j), c_{ij} represents its link capacity. $X_{ij}^k \in [0,1]$ is the percentage



(a) Average FCT of all flows

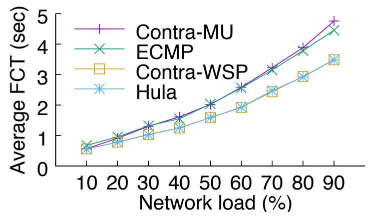


(b) Small flows (<= 100 KB)

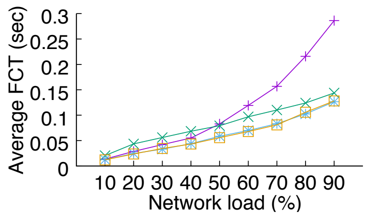


(c) Large flows (>= 10 MB)

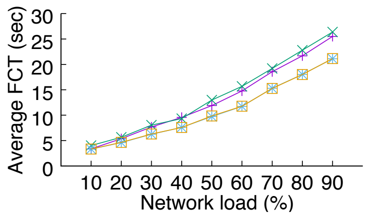
Figure 20: FCT results for web search workload in 6-ary Fattree topology.



(a) Average FCT of all flows

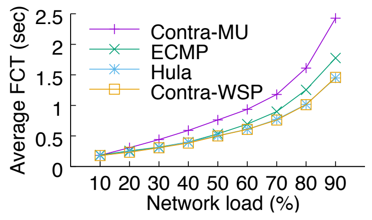


(b) Small flows (<= 100 KB)

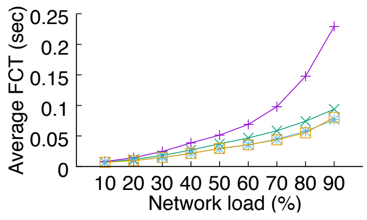


(c) Large flows (>= 10 MB)

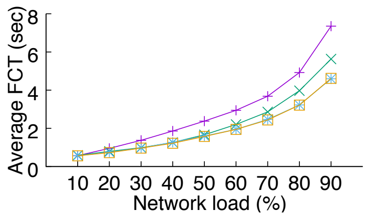
Figure 21: FCT results for web search workload in asymmetric 6-ary Fattree topology.



(a) Average FCT of all flows

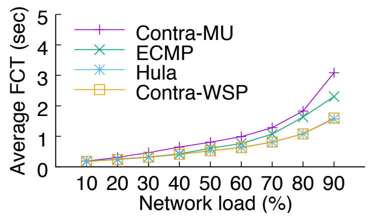


(b) Small flows (<= 100 KB)

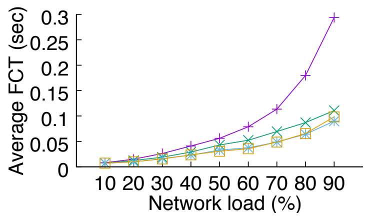


(c) Large flows (>= 1 MB)

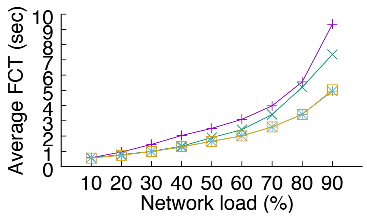
Figure 22: FCT results for cache workload in 6-ary Fattree topology.



(a) Average FCT of all flows

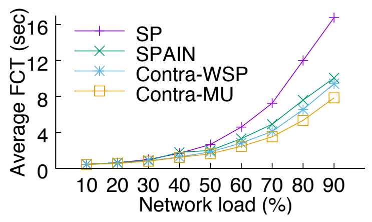


(b) Small flows (<= 100 KB)

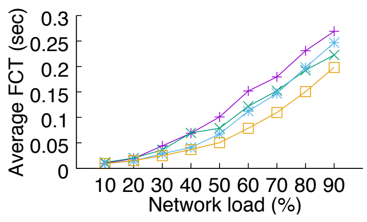


(c) Large flows (>= 1 MB)

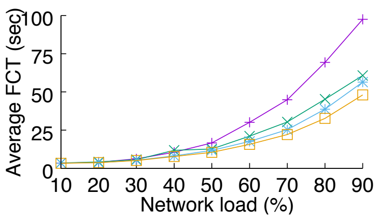
Figure 23: FCT results for cache workload in asymmetric 6-ary Fattree topology.



(a) Average FCT of all flows

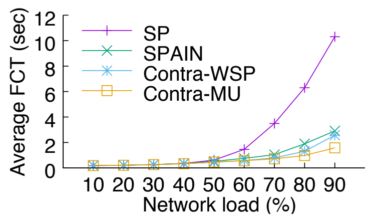


(b) Small flows (<= 100 KB)

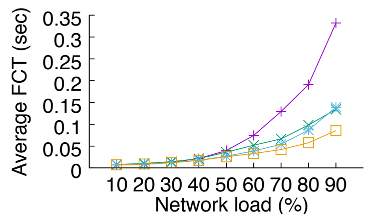


(c) Large flows (>= 10 MB)

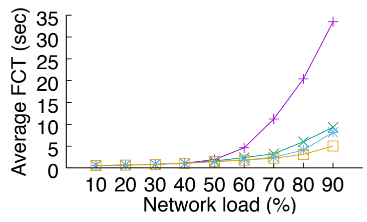
Figure 24: FCT results for web search workload in Abilene topology.



(a) Average FCT of all flows



(b) Small flows (<= 100 KB)



(c) Large flows (>= 1 MB)

Figure 25: FCT results for cache workload in Abilene topology.

of traffic a solution sends to link (i,j) for a given commodity flow from the source s_k to the destination t_k, where $k \in K$ represents a commodity flow chosen from the set K of flows to be sent. The total demand for a flow k is d_k. Our goal is to minimize the maximum link utilization $\alpha \in [0,1]$ across the network.

The problem can then be formulated as follows:

$$\min(\alpha) \tag{1}$$

s.t.

$$\sum_{j:(i,j)\in E} X_{ij}^k - \sum_{j:(j,i)\in E} X_{ji}^k = 0, \qquad k \in K, i \neq s_k, t_k \tag{2}$$

$$\sum_{j:(i,j)\in E} X_{ij}^k - \sum_{j:(j,i)\in E} X_{ji}^k = 1, \qquad k \in K, i = s_k \tag{3}$$

$$\sum_{k\in K} d_k X_{ij}^k \leq c_{ij}\alpha, \qquad (i,j) \in E \tag{4}$$

$$0 \leq X_{ij}^k \leq 1, \alpha \geq 0.$$

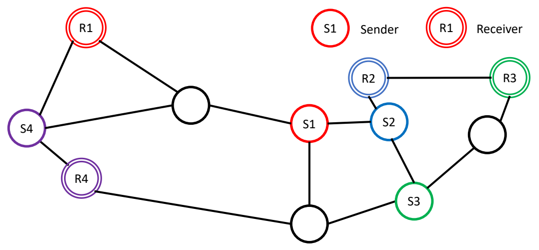
Equations 1 and 4 define α as the maximum link utilization and set the objective function to minimize this. Equation 2 encodes the flow conservation principle, which specifies that all nodes should have the same amount of incoming and outgoing traffic, except for the sources and destinations. Equation 3 specifies the source switch of each flow.

Simplifying assumptions: We note that this formulation makes several simplifying assumptions when testing the systems. In order to ensure that the created demands are static, we used UDP instead of TCP to avoid its flow control algorithm, and we artificially made the buffers deep enough to avoid packet loss. Given that the FMCF formulation does not have the notion of flowlets (which requires reasoning with timing behaviors), we have configured ECMP, SPAIN, and Contra to perform per-packet load balancing to emulate the problem that FMCF models. This configuration significantly disadvantages Contra, because unlike ECMP, SPAIN, which are inherently multipath, Contra is designed to spread traffic per-flowlet over time, and it only changes paths based on periodic probes. Since this feature is disabled, we instead measured the utilization of all systems at a coarser timescale over multiple RTTs, so that Contra is given an opportunity to balance the load. Despite the above simplifications, we believe that the results we obtain are still illustrative, as these assumptions make it possible to derive an optimal solution that we can compare the actual systems against.

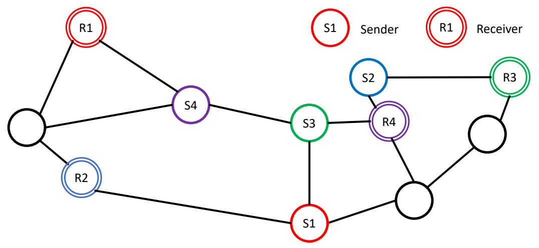
G.2 Experimental results for FMCF

Figures 27 and 28 show three setups where the optimal solutions returned by our solver are 20%, 40%, and 60%, respectively. For each setup, we have tested the systems on a Fattree topology and on Abilene. On a Fattree, both ECMP and Contra are very close to the optimum: they are 0.049% and 0.16% higher than optimum on average. Since ECMP splits traffic on a per-packet basis, it is expected to achieve almost perfect load balancing; Contra underperforms slightly since it only changes forwarding decisions based on periodic probes, but it performs close to ECMP and the optimum.

On a general topology, the performance of SPAIN is highly dependent on the locations of senders and receivers, as its load balancing mechanism precomputes non-overlapping paths when possible. We found that, when alternative paths in SPAIN do not overlap, it performs very close to optimum (worse only by 2.53%), but when paths overlap, SPAIN could underperform by as much as 6.55%. Contra, on the other hand, has consistent performance, and achieves similar performance with the same scenarios used to evaluate SPAIN. Compared to the optimum, the results for Contra are 2.41% and 2.36% higher, respectively. Figure 26 shows the two setups we have used for the experiments with SPAIN.



(a) Abilene setup 1



(b) Abilene setup 2

Figure 26: Selection of sender/receiver pairs on Abilene.

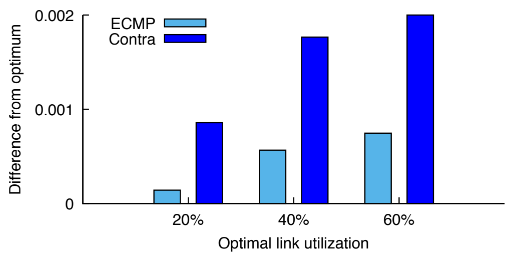


Figure 27: The performance of Contra is close to the optimum in the FMCF formulation (Fattree topology).

H Waypoint policy

The performance evaluation in our main paper has focused on policies that do not involve regular expressions, because regular expressions constrain paths rather than optimize for performance. Nevertheless, we have conducted a set of ex-

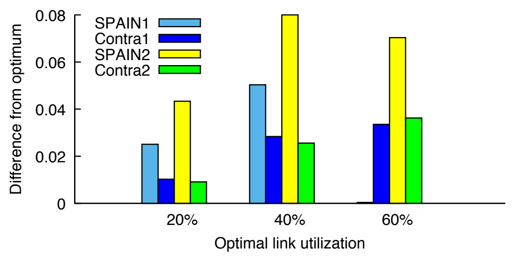


Figure 28: The performance of Contra is close to the optimum in the FMCF formulation (Abilene topology).

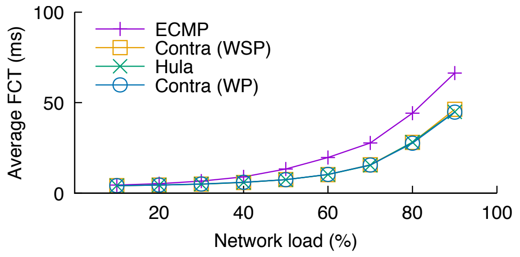


Figure 29: FCT for the waypoint policy (workload: web search)

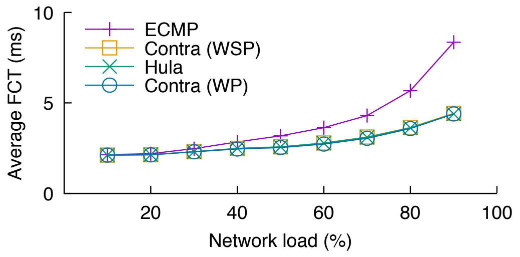


Figure 30: FCT for the waypoint policy (workload: cache)

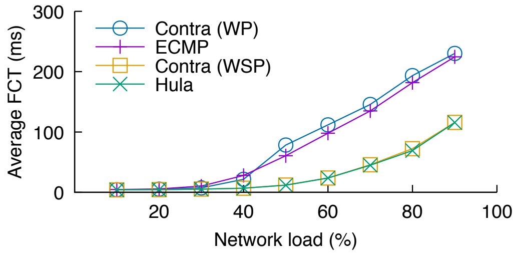


Figure 31: FCT for the waypoint policy on an asymmetric topology (workload: web search)

periments on such policies, and report the key findings in this section. We have used the waypoint policy (WP) with one regular expression, and measured the performance and protocol overhead in different workloads.

Flow completion time. Figures 29 and 30 show the FCT achieved by the WP policy on a symmetric data center topology, on the web search and cache datasets, respectively. As we can see, on the symmetric topology, WP performs similarly to Hula and WSP on both workloads. Figures 31 and 32 show the FCT results for the asymmetric topology, where we have injected a failed link. When the topology is asymmetric, WP performs worse than ECMP. This is expected, as WP imposes additional path constraints.

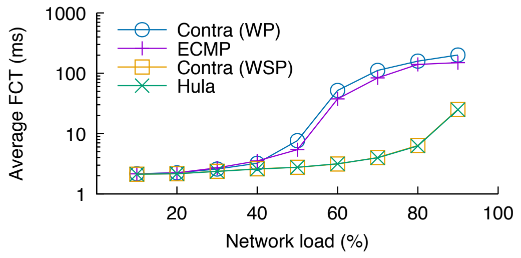


Figure 32: FCT for the waypoint policy on an asymmetric topology (workload: cache)

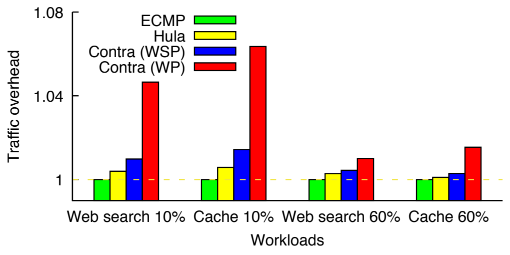


Figure 33: The tags in the waypoint (WP) policy introduce more traffic overhead.

Protocol overhead. Figure 33 shows the traffic overhead of the WP policy. As we can see, WP sends more traffic than WSP because it tags packets with policy states and creates separate probes for different regular expression matches. At 10% load, 92% of the traffic overhead is due to probes and 8% due to tags; at 60% load, 70% of the traffic overhead is due to probes and 30% due to tags.

I Traffic overhead

To evaluate the traffic overhead incurred by Contra due to additional probes (the policies below do not require tags), we measured the amount of traffic sent over the network by Contra, Hula, and ECMP at 10% and 60% network load. Figure 34 shows the results a normalized by the traffic sent by ECMP (i.e., no extra tags or probes). Across workloads, Contra incurred 0.79% more traffic than ECMP, and 0.44% more than Hula, which seems to be reasonable.

We also evaluated the traffic overhead of SPAIN and Contra on the Abilene network, at 10% and 60% network load. Figure 35 shows the results. We found that only 0.54% of traffic is due to the extra probes in Contra. Interestingly, al-

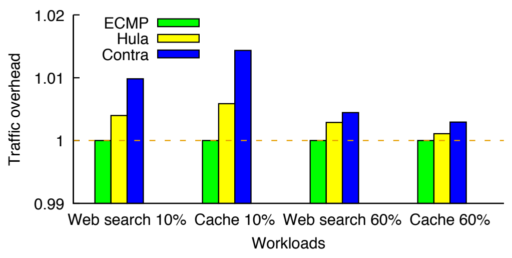


Figure 34: The traffic overhead of Contra is low.

though SPAIN did not use any extra probes, the amount of traffic SPAIN sent across the network is higher than that of Contra, and even higher than the total traffic of Contra. This is because SPAIN's paths are on average longer than these used in Contra. As a result, Contra requires 6.65% less network bandwidth than SPAIN.

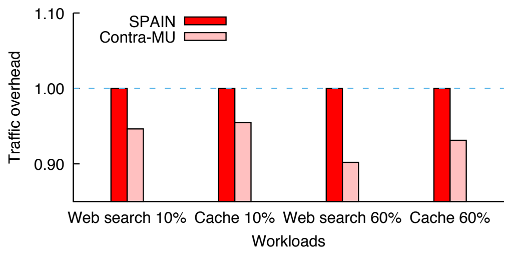


Figure 35: The traffic overhead of Contra and SPAIN on the Abilene network

FLAIR: Accelerating Reads with Consistency-Aware Network Routing

Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, Samer Al-Kiswany
University of Waterloo, Canada

Abstract

We present FLAIR, a novel approach for accelerating read operations in leader-based consensus protocols. FLAIR leverages the capabilities of the new generation of programmable switches to serve reads from follower replicas without compromising consistency. The core of the new approach is a packet-processing pipeline that can track client requests and system replies, identify consistent replicas, and at line speed, forward read requests to replicas that can serve the read without sacrificing linearizability. An additional benefit of FLAIR is that it facilitates devising novel consistency-aware load balancing techniques.

Following the new approach, we designed FlairKV, a key-value store atop Raft. FlairKV implements the processing pipeline using the P4 programming language. We evaluate the benefits of the proposed approach and compare it to previous approaches using a cluster with a Barefoot Tofino switch. Our evaluation indicates that, compared to state-of-the-art alternatives, the proposed approach can bring significant performance gains: up to 42% higher throughput and 35-97% lower latency for most workloads.

1. Introduction

Replication is the main reliability technique for many modern cloud services [1, 2, 3] that process billions of requests each day [3, 4, 5]. Unfortunately, modern strongly-consistent replication protocols [6] – such as multi-Paxos [7], Raft [8], Zab [9], and Viewstamped replication (VR) [10] – deliver poor read performance. This is because these protocols are leader-based: a single leader replica (or leader, for short) processes every read and write request, while follower replicas (followers for short) are used for reliability only.

Optimizing read performance is clearly important; for instance, the read-to-write ratio is 380:1 in Google's F1 advertising system [11], 500:1 in Facebook's TAO [5], and 30:1 in Facebook memcached deployments [12]. Previous efforts have attempted to accelerate reads by giving read leases [13] to some [14] or all followers [1, 15, 16], While holding a lease, a follower can serve read requests without consulting the leader; each lease has an expiration period. Unfortunately, this approach complicates the system's design, as it requires careful management of leases, affects the write operation – as all granted leases need to be revoked before an object can be modified – and imposes long delays when a follower holding a lease fails [1, 14].

Alternatively, many systems support a relaxed consistency model (e.g., eventual [2, 17, 18, 19, 20, 21] or read-your-write [5, 21, 22]), in exchange for the ability to read from followers, albeit the possibility of reading stale data.

In this paper, we present the fast, linearizable, network-accelerated client reads (FLAIR), a novel protocol to serve reads from follower replicas with minimal changes to current leader-based consensus protocols without using leases, all while preserving linearizability. In addition to improving read performance, FLAIR improves write performance by reducing the number of requests that must be handled by the leader and employing consistency-aware load-balancing.

FLAIR is positioned as a shim layer on top of a leader-based protocol (§3). FLAIR assumes a few properties of the underlying consensus protocol: the operations are stored in a replicated log; at any time, there is at most one leader in the system that can commit new entries in the log; reads served by the leader are linearizable; and after committing an entry in the log, the leader knows which followers have a log consistent with its log up to that entry. These properties hold for all major leader-based protocols (Raft [8], VR [10], DARE [23], Zookeeper [2], and multi-Paxos [24, 25, 26]).

FLAIR leverages the power and flexibility of the new generation of programmable switches. The core of FLAIR is a packet-processing pipeline (§4) that maintains compact information about all objects stored in the system. FLAIR tracks every write request and the corresponding system reply to identify which objects are stable (i.e., not being modified) and which followers hold a consistent value for each object, then uses this information to forward reads of stable objects to consistent followers. Followers optimistically serve reads and the FLAIR switch validates read replies to detect stale values. If the switch suspects that a reply from a follower is stale, it will drop the reply and resubmit the read request to the leader.

An additional benefit of FLAIR is that it facilitates the building of novel consistency-aware load balancing techniques. In systems that grant a lease to followers [1, 14, 15, 16], clients send read requests to a randomly selected follower. If the follower does not hold a lease, it blocks the request until it obtains a lease, or it forwards the request to the leader; either way, this approach adds additional delay. FLAIR does not incur this inefficiency as FLAIR load balances read requests only among followers that hold a consistent value for the requested object. In this paper we design three consistency-aware load balancing techniques (§6): random, leader avoidance, and load awareness.

Unlike other systems that use switch's new capabilities [27, 28, 29], FLAIR does not rely on the controller to update the switch information after every write operation, as this approach would add unacceptable delays. Instead, FLAIR piggybacks control messages on system replies, and the switch extracts and processes them.

Despite its simplicity, implementing this approach is complicated by the limitations of programmable switches (§2) and the complexity of handling switch failures, network partitioning, and packet loss and reordering (§4).

To demonstrate the powerful capabilities of the proposed approach, we prototyped FlairKV (§6), a key-value store built atop Raft [8]. We made only minor changes to Raft's implementation [30] to enable followers to serve reads, make the leader order write requests following the sequence numbers assigned by the switch, and expose leader's log information to the FLAIR layer. The packet-processing pipeline was implemented using the P4 programming language [31]. We implemented the three aforementioned load-balancing techniques (§6).

Our evaluation of FlairKV (§7) on a cluster with a Barefoot Tofino switch shows that FLAIR can bring sizable performance gains without increasing the complexity of the leader-based protocols or the write operation overhead. Our evaluation with different read-to-write ratios and workload skewness shows that FlairKV brings up to 2.8 times higher throughput than an optimized Raft implementation, at least 4 times higher throughput compared to Viewstamped replication, Raft, and FastPaxos, and up to 42% higher throughput and up to 35-97% lower latency for most workloads compared to state-of-the-art leases-based design [1, 16].

The performance and programmability of the new generation of switches opens the door for the switches to be used beyond traditional network functionalities. We hope our experience will inform a new generation of distributed systems that co-design network protocols with systems operations.

2. Background

In this section, we present an overview of leader-based consensus protocols, followed by a look at the new programmable switches and their limitations.

2.1. Leader-based Consensus

Leader-based consensus (LC) protocols [8, 9, 10, 23, 24, 25] are widely adopted in modern systems [2, 3, 4, 16]. The idea of having a leader that can commit an operation in a single round trip dates back to the early consensus protocols [7, 32]. Having a leader reduces contention and the number of messages, which greatly improves performance [7, 24].

LC protocols divide time into terms (a.k.a. views or epochs). Each term has a single leader; if the leader fails, a new term starts and a new leader is elected.

Clients send write requests to the leader (1 in Figure 1). The leader appends the request to its local log (2) and then sends the request to all follower replicas (3). A follower appends the request to its log (4) before sending an acknowledgment to the leader (5). If the leader receives an acknowledgment from a majority of its followers, the operation is considered *committed*. The leader *applies* the operation to its local state machine (e.g., in memory key-value store in Figure 1) in (6), then acknowledges the operation to the client (7). The leader will asynchronously inform the followers that it committed the operation. Followers maintain a *commit_index*, a log index pointing to the last committed operation in the log; when a follower receives the commit notification, it advances its *commit_index* and applies the write to its local store.

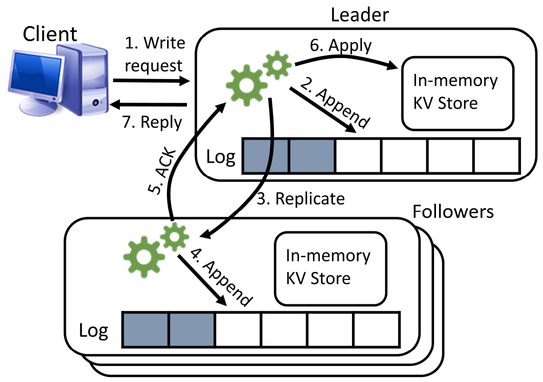


Figure 1. The path for a write operation.

The replicated log has two properties that make it easy to reason about: it is guaranteed that if an operation at index *i* is committed, then every operation with an index smaller than *i* is committed as well; and if a follower accepts a new entry to its log, it is guaranteed that its log is identical to the leader's log up to that entry.

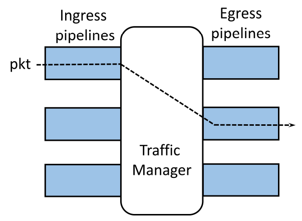
Client read requests are also sent to the leader. In Raft, the leader sends a heartbeat to all followers to make sure it is still the leader. If a majority of followers reply, the leader serves the read form its local store: it will check that all committed operations related to the requested object are applied to the local store before serving the request.

A common optimization is the leader lease optimization. Instead of collecting a majority of heartbeats for every read request, a majority of the followers can give the leader a lease [8, 24]. While holding a lease, the leader serves reads locally without contacting followers. Unfortunately, even with this optimization, the performance of the leader-based protocols is limited to a single-node performance.

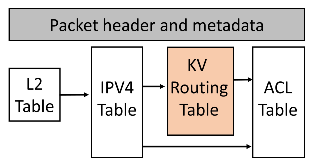
2.2. Programmable Switches

Programmable switches allow the implementation of an application-specific packet-processing pipeline that is deployed on network devices and executed at line speed. A number of vendors produce network-programmable ASICs, including Barefoot's Tofino [33], Cavium's XPliant [34], and Broadcom Trident 3 [35].

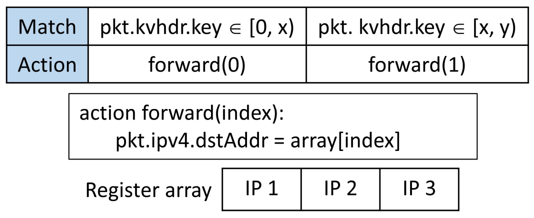
Figure 4(a) illustrates the basic data plane architecture of modern programmable switches. The data plane contains three main components: ingress pipelines, a traffic manager,



(a) Switch data plane.



(b) Pipeline for routing based on a hash-based key



(c) Simple match-action stage for routing based on a hash-based key for the KV routing table in subfigure (b)

Figure 4. Switch data plane.

and egress pipelines. A packet is first processed by an ingress pipeline before it is forwarded by the traffic manager to the egress pipeline that will finally emit the packet.

Each pipeline is composed of multiple stages. At each stage, one or more tables match fields in the packet header or metadata; if a packet matches, the corresponding action is executed. Programmers can define custom headers and metadata as well as custom actions. Each stage has its own dedicated resources, including tables and register arrays (a memory buffer). Figure 4(b) shows a simple example of a pipeline that routes a request to a key-value store based on the key, and Figure 4(c) shows the details of the KV routing stage. The stage forwards the request based on the key in the packet's custom L4 header. The programmer implements a forward() action that accesses the register array holding nodes' IP addresses. An external controller can modify the register array and the table entries.

Stages can share data through the packet header and small per-packet metadata (a few hundred bytes in size) that is propagated between the stages as the packet is processed throughout the pipeline (Figure 4(b)). The processing of packets can be viewed as a graph of match-action stages.

Programmers use domain-specific languages like P4 [36] to define their own packet headers, define tables, implement custom actions, and configure the processing graphs.

Challenges. While programmable ASICs and their domain-specific languages significantly increase the flexibility of network switches, the need to execute custom actions at line speed restricts what can be done. To process packets at line speed, P4 and modern programmable ASICs have to meet strict resource and timing requirements. Consequently, modern ASICs limit (1) the number of stages per pipeline, (2) the number of tables and registers per stage, (3) the number of times any register can be accessed per packet, (4) the amount of data that can be read/written per-packet per register, (5)

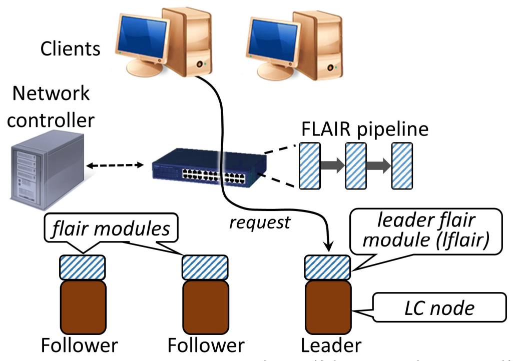


Figure 2. System architecture. The solid arrow shows a client request, while the dashed arrow show control messages.

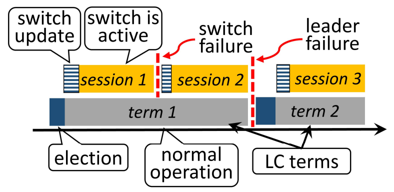


Figure 3. FLAIR sessions. Time is divided into terms. Each term starts with a leader election. Each term has one or more sessions that start with updating the switch data.

the size of per-packet metadata that is passed between stages. Finally, modern ASIC's lack support of loops or recursion.

3. FLAIR Overview

FLAIR is a novel protocol that targets deployments in a single data center. Figure 2 shows the system architecture, which consists of a programmable switch, a central controller, and storage nodes. Typically, multiple FLAIR instances are deployed with each serving a disjoint set of objects. For simplicity, we present a FLAIR deployment with one replica set (i.e., one leader and its followers).

FLAIR is based on the following assumptions; the network is unreliable and asynchronous, as there are no guarantees that packets will be received in a timely manner or even delivered at all, and there is no limit on the time a node or switch takes to process a packet. Finally, FLAIR assumes a non-byzantine failure model in which nodes and switches may stop working but will never send erroneous messages.

FLAIR divides time into sessions (Figure 3). During a session the leader is bonded to a single switch that runs the FLAIR pipeline. Each session has a unique id that is assigned in a strictly increasing order. A session ends when a leader fails or the leader suspects that the switch has failed. An LC term may have one or more sessions, but a session does not span multiple terms.

A session starts with the FLAIR module at the leader (dubbed the *lflair* module) incrementing the session id, committing it to the LC log, updating the switch information about the objects in the system, then activating the session at the switch. *lflair* module keeps the switch's information up to date while in an active session. If the switch does not have an active session it drops all FLAIR packets.

Clients. FLAIR is accessed through a client library with a simple read/write/delete interface. Read (get) and write (put) read or write entire objects. The library adds a special FLAIR packet header to every request, that contains an operation code (e.g., read) and a key (a hash-based object identifier).

Controller. Our design targets data centers that use a SDN network following a variant of the multi-rooted tree topology [37, 38]. A central controller uses OpenFlow [39] to manage the network by installing per-flow forwarding, filtering, and rewriting rules in switches.

As with previous projects that leverage SDN capabilities [27, 29, 40, 41], the controller assigns a distinct address for each replica set. The controller installs forwarding rules to guarantee that every client request for a range of keys served by a single replica set is passed through a specific switch (dubbed FLAIR switch); that switch will run the FLAIR logic for that range of keys. The controller typically selects a common ancestor switch of all replicas and installs rules to forward system replies through the same switch. Only client request/replies are routed through the FLAIR switch, leader-follower messages do not have the FLAIR header nor are necessarily routed through the FLAIR switch.

While this approach may create a longer path than traditional forwarding, the effect of this change is minimal. Li et al. [40] reported that for 88% of cases, there is no additional latency, and the 99th percentile had less than 5 μs of added latency. This minimal added latency is due to the fact that the selected switch is the common ancestor of target replicas and client packets have to traverse that switch anyway.

On a switch failure, the controller selects a new switch and updates all the forwarding rules accordingly. The controller load balances the work across switches by assigning different replica sets to different switches.

Storage Nodes. The storage nodes run the FLAIR and LC protocols. For read requests, before serving a read, followers verify that all committed writes to the requested object have been applied to the follower's local storage.

Write requests are processed by the leader. After a successful write operation, the leader passes to the *lflair* module the log index at which the write was committed and the list of followers that accepted the write operation and have a consistent log up to that log index. The *lflair* encodes this list into a compact bitmap and uploads it and the log index to the switch (piggybacked on the write reply).

Programmable Switch. The switch is a core component of FLAIR: it tracks every write request and the corresponding reply to identify which objects are stable (not being modified) and which replicas have a consistent value of each object (encoded in the bitmap provided by the *lfair* module). If a read is issued while there are outstanding writes for the target object (i.e., writes without corresponding replies), the read is forwarded to the leader. If a read request is processed by the switch when there are no outstanding writes to the requested object, the switch forwards the request to one of the followers included in the last bitmap for the object sent by the *lflair* module. Followers optimistically serve read requests. The switch inspects every read reply; if it suspects that a follower returned stale data (Section 4.4), it will conservatively drop the reply and forward the request to the leader. FLAIR forwards all writes to the leader.

FLAIR also includes techniques to handle multiple concurrent writes to the same object (Section 4.3), packets reordering (Section 4.6), and tolerating switch, node, and network failures (Section 4.6).

4. System Design

4.1. Network Protocol

Packet format. FLAIR introduces an application-layer protocol embedded in the L4 payload of packets. Similar to many other storage systems [27, 29, 40], FLAIR uses UDP to issue client requests in order to achieve low latency and simplify request routing. Communication between replicas uses TCP for its reliability. A special UDP port is reserved to distinguish FLAIR packets; for UDP packets with this port, the switch invokes the FLAIR custom processing pipeline. Other switches do not need to understand the FLAIR header and will treat FLAIR packets as normal packets. In this way, FLAIR can coexist with other network protocols.

Figure 5 shows the main fields in the FLAIR header. We briefly discuss the fields here (a detailed discussion of the protocol is presented next):

- `OP`: the request type. Clients populate this field in the request packet (e.g., read, or write); replicas populate this field in the reply packets (e.g., read_reply, write_reply).
- `KEY`: hash-based object identifier.
- `SEQ`: a sequence number added by the switch. The switch increments the sequence number on every write operation.
- `SID`: a unique session id. The <`SID`, `SEQ`> combination represents a unique identifier for every write request.
- `LOG_IDX`: a log index. In a write_reply, the log index indicates the index at which the write was committed. For reads, the switch populates `LOG_IDX` to make sure the followers' logs are committed and applied up to that index.
- `CFLWRS`: In write_reply, the `CFLWRS` is a map of the followers that have a consistent log up to `LOG_IDX`.

Following the FLAIR header is the original LC protocol payload, which includes the value for read/write operations.

4.2. Switch Data Structures

To process a read request, the switch performs two specific tasks (Section 4.4). First, it forwards read requests to consistent followers while balancing the load among them. Second, it verifies the read replies to preserve safety. To perform these tasks, the switch maintains two data structures: a session array and a key group array.

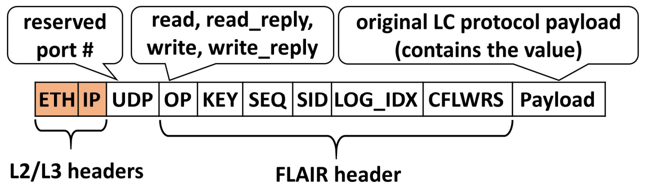

reserved port #
read, read_reply, write, write_reply
original LC protocol payload (contains the value)
ETH
IP
UDP
OP
KEY
SEQ
SID
LOG_IDX
CFLWRS
Payload
L2/L3 headers
FLAIR header

Figure 5. FLAIR packet format.

```
SessionArrayEntry {                 KGroupArrayEntry {
  bit<1>  is_active;                  bit<1>  is_stable;
  bit<32> session_id;                 bit<64> seq_num;
  bit<32> leader_ip;                  bit<64> log_idx;
  bit<64> session_seq_num;            bit<8>  consistent_followers;
  bit<48> heartbeat_tstamp; }       }
```

Listing 1. Session and kgroup entries. The numbers indicate the field size in bits.

Session array. A single switch typically supports multiple replica sets (i.e., FLAIR+LC instances) with each set storing a disjoint set of keys. Each entry in the session array maintains the session status for a single replica set. An entry contains an is_active flag, session id, leader IP address, current session sequence number, and the timestamp of the last heartbeat received from the *lflair* module (Listing 1). When is_active is true, we say the session is *active*, which indicates that the session entry and kgroup array are consistent with the leader's information. The switch processes packets using the FLAIR custom pipeline only if the session is active; otherwise, it will drop all FLAIR packets, rendering the system unavailable to clients until the switch can reach the *lflair* module and sync its session entry and key group array.

Key group (KGroup) array. To decide if followers can serve a certain read request, the switch needs to maintain information about which followers have the latest committed value of every object. Maintaining such information in the switch ASIC's memory is not feasible; instead, FLAIR groups objects based on their key and maintains aggregate information per group. We use the most significant *k* bits of the key to map an object to a key group (kgroup).

Every FLAIR+LC instance has a dedicated kgroup array. Each entry in the array (Listing 1) contains the status of a single kgroup, including an is_stable flag that indicates if all objects in the kgroup are stable. If a kgroup is not stable (is_stable is false), this indicates that at least one object in the kgroup is being modified (i.e., has an outstanding write in the system). The array entry also includes the sequence number (seq_num) of the last write request processed by the switch for any object in the kgroup, the log index (log_idx) of the last successful write to any object in the kgroup, and the consistent_followers bitmap pointing to all followers that have a consistent log up to log_idx.

4.3. Handling Write Requests

To issue a write request, a client populates the OP and KEY fields of the FLAIR packet header and puts the value in the payload, then sends the request.

When the switch receives the request, it will mark the corresponding kgroup entry as unstable. The switch will increment the session_seq_num in the session array and use it to populate the sequence number (seq_num) in the kgroup entry and the sequence number (SEQ) in the request header. Finally, the switch populates the session id (SID) field in the header and forwards the request to the leader.

The *lflair* module will verify that the session id is valid, and will pass the write request to the leader. The leader verifies that the <SID, SEQ> combination is larger than the <SID, SEQ> number of any previous write request it ever received, else it will drop the packet. The LC leader will process the write request following the LC protocol (Section 2.1): it will replicate the request to all followers, and when a majority of followers acknowledge the operation, the write operation is considered committed. A follower will acknowledge a write operation only if its log is identical to the leader's log up to that entry.

For the write reply, the leader will pass the following to the *lflair* module: the LC protocol payload for the write_reply, the log index at which the write was committed, and the list of followers that acknowledged the write. The *lflair* module will create the write reply packet with the leader provided payload, and will populate the LOG_IDX and the bitmap of the consistent followers (CFLWRS) using the information provided by the leader. *lflair* module populates the sequence number (SEQ) in the write_reply header using the SEQ of the corresponding write request. The *lflair* module then sends the write_reply packet.

The switch will process the write_reply header and verify its session id. The switch will compare the sequence number (SEQ) of the reply to the sequence number (seq_num) in the kgroup entry; if they are equal, this signifies that no other write is concurrently being processed in the system for any object in the kgroup. Consequently, it will update the log_idx and the consistent_followers fields in the kgroup entry using the values in the write reply. Then it will mark the kgroup stable and forward the reply to the client.

If the sequence number in the reply is smaller than the sequence number in the kgroup entry, this indicates that a later write to an object in the same kgroup has been processed by the switch. In this case, the switch forwards the write reply to the client without modifying the kgroup entry. The kgroup entry remains unstable until the last write to the kgroup (with a SEQ number in the write_reply equal to the seq_num in the kgroup entry) is acknowledged by the leader.

In a nutshell, the switch acts as a look-through metadata cache. Write requests invalidate the switch metadata related to the accessed kgroup, and write replies update the kgroup metadata at the switch. As we see next, the kgroup metadata is used to consistently load balance reads.

4.4. Handling Read Requests

Clients fill the OP and KEY fields of the FLAIR header and send the request. When the switch receives the request, it will check the kgroup entry. If the entry is stable, the switch will fill the sequence number (SEQ) and log index (LOG_IDX) header fields using the values in the kgroup entry. Then it will forward the request to one of the followers indicated in the consistent_followers bitmap. Section 6.2 details our load balancing techniques.

If the kgroup entry is not stable, the switch forwards the read request to the leader. We note that there is a chance for false positives in this design, as a single write will render all the objects in the same kgroup unstable. This is a drawback of maintaining information per group of keys. This inefficiency is incurred by leases-based protocols as well, as they maintain a lease per group of objects.

When a follower receives a read request, the follower's FLAIR module validates the request, then calls advance_then_read(LOG_IDX, key) routine, which compares the follower's commit_index to LOG_IDX. If the commit_index is smaller, the follower advances its commit_index to equal LOG_IDX, apply all the log entries to the local store, then serve the read request. The FLAIR module will populate the read_reply header; for the SEQ and SID fields, it will use the values found in the read request header.

We note that it is safe to advance the follower's commit_index to match the LOG_IDX in the read request, as the switch forwards read requests to a follower only if the leader indicates that all entries in the log up to that log index are committed, and that this specific follower is one of the replicas that have a log consistent to the leader's log up to that index. We discuss FLAIR correctness in Section 5.

When the switch receives a read_reply from a follower, it validates the session id, then verifies that the SEQ number of the read_reply equals the seq_num of the kgroup entry. If the sequence numbers are not equal, this signifies that a later write request was processed by the switch and there is a chance the follower has returned stale value. In this case, the switch drops the read_reply, generates a new read request using the KEY field from read_reply packet, and submits the read request to the leader. If the sequence number of the read_reply equals the sequance number in the kgroup entry, the switch forwards the reply to the client.

If a read request is forwarded to the leader, the *lflair* module verifies the session id, then calls advance_then_read(LOG_IDX, key). The switch verifies that the leader reply is valid (i.e., has the correct session id) before forwarding it to the client, without checking the seq_num in the kgroup entry.

4.5. Session Start Process

On the start of a new session, the *lflair* module reads the last session id from the LC log, increments it, and commits the new session id to the LC log. Then the *lflair* module asks the central controller for a new switch. The central controller neutralizes the old switch (making it drop all FLAIR packets) and reroutes FLAIR packets to a new switch, then confirms the switch change to the *lflair* module. This step guarantees that at any time at most one FLAIR switch is active. The *lflair* module updates the session entry (Listing 1) at the switch with the current leader IP and session id. For each new session, session_seq_num is reset to zero.

Populating the kgroup array. The *lflair* module maintains a copy of the kgroup array similar to the one maintained by the switch. If the leader did not change between sessions (e.g., the session change is due to switch failure), the kgroup array at the *lflair* module is up to date. The *lflair* module will set the seq_num entry in all kgroup entries to zero (equal to the session_seq_num in the session entry)., and upload it to the switch.

If the kgroup array at the *lflair* module is empty – for instance, after electing a new leader – the *lflair* module will query the leader for three pieces of information: its commit_index, the list of followers with the same commit_index, and a list of all uncommitted operations in the log (i.e., the operations after the commit_index in the log). The list of uncommitted operations is typically small, as it only includes operations that were received before the end of the last term but were not committed yet. The *lflair* module will traverse the list of uncommitted writes and mark their target kgroup entries unstable. For all other kgroup entries, the *lflair* module will mark them stable and set their seq_num to zero, log_idx to the leader's commit_index, and consistent_followers to include all the followers that have the same commit_index as the leader's. After updating the session entry and the kgroup array at the switch, the *lflair* module activates the switch session (sets is_active to true).

4.6. Fault Tolerance

Follower Failure. We rely on the LC protocol to handle follower failures. To avoid sending read requests to a failing follower, the leader notifies the *lflair* module when it detects the failure of a follower. The *lflair* module removes the follower from the switch-forwarding table (Section 3).

Leader Failure. On leader failure, a new leader is elected and a new term starts. The new leader informs the *lflair* module of the term change; and the *lflair* module starts a new session (Section 3).

The *lflair* module sends periodic heartbeats to the switch. Upon receiving a heartbeat, the switch determines whether it is from the current session. If the heartbeat is valid, the switch updates the heartbeat_timestamp in the session array and replies to the *lflair* module.

Switch Failure. If the *lflair* module misses the switch heartbeats for a *switch_stepdown* period of time (3 heartbeats in our prototype), the *lflair* module will suspect that the switch has failed and will start a new session. For efficiency (i.e. does not affect safety), if the switch misses three heartbeats from the leader, it will deactivate the session.

Network Partitioning. If a network partition isolates the switch from the leader, the leader treats it as a failed switch, as detailed above. If a network partition isolates the switch from a follower, read requests forwarded to the follower will time out and the client will resubmit the request. This failure affects performance, but not correctness. Upon determining that a follower is not reachable, the leader removes it from the forwarding table, as in the case of the failed follower described above.

Packet Loss. If a read or write request is lost, the client times out and resubmits the request. If a write reply is lost before reaching the switch, the kgroup entry will remain unstable until a new write operation to any key in the kgroup succeeds. While the kgroup entry is not stable, all read requests are forwarded to the leader.

Packet Reordering. It is critical for FLAIR correctness that the leader processes write requests in the same order that they are processed by the switch. Every write operation gets a unique <SID, SEQ> number. The switch marks a kgroup entry unstable until the leader replies to the last write issued for a key in the kgroup. Consequently, if the leader processes the requests out of order, the switch will incorrectly mark a kgroup stable while the out-of-order writes are modifying its objects. To prevent this scenario, the leader keeps track of the largest <SID, SEQ> it has ever processed and drops any write request with a smaller number. While session numbers (SIDs) are maintained in the log, the largest processed sequence number is retained in memory. If the leader fails, the new leader starts a new session, increments the session id (SID), and sets the session sequence number (SEQ) to zero.

5. Correctness

FLAIR guarantees linearizability, which means that concurrent operations must appear to be executed by a single machine. FLAIR relies on the LC protocol for any operation that updates the log and for reads from the leader.

FLAIR only adds the ability to serve reads from followers. In this section, we sketch out the proof of FLAIR correctness when the read is served by a follower. A full and detailed proof is available in the technical report [42]. Further, we used the TLA+ model checking tool [43] to verify the FLAIR correctness. We started from Raft's TLA+ specification [44] and extended it with a formal specification for our protocol and new invariants to validate the linearizability of reads. The TLA+ specification is in our technical report [42].

Safety. FLAIR guarantees that all read replies are linearizable. FLAIR trusts that the leader's read replies are linearizable and forwards them to the client. For reads served by followers, FLAIR guarantees that the read reply returns an identical value, as if the read was served by the leader. This is guaranteed using the following two steps:

First, when the switch receives a read request, the switch forwards that request to followers only when the switch has an active session and the kgroup entry is stable. This signifies that the switch information is up-to-date with the *lflair* module's information. Identifying a kgroup entry as stable signifies that there are no current writes to any object in the kgroup and that the last leader-provided consistent_followers bitmap points to followers that have the last committed value for every object in the kgroup. Consequently, any of the consistent followers will return a value identical to the leader's value.

Second, after forwarding a read request to a follower (say, flwrA), the switch may receive a write request that modifies the object. The leader may replicate the write request to a majority of nodes that does not include flwrA. If the leader processes the write request before flwrA serves the read request, flwrA will return stale data. To avoid this case, the switch performs a safety check on every read reply coming from followers: it verifies that the kgroup is still stable, and that the sequence number in the read_reply is equal to the sequence number in the kgroup entry. If the sequence numbers do not match (which indicates that there are later writes to objects in the kgroup), the switch conservatively drops the read reply and forwards the request to the leader. *At all times, reads are linearizable in FLAIR.*

6. Implementation

To demonstrate the benefits of the new approach, we prototyped FlairKV, a FLAIR-based key-value store built atop Raft [30]. We chose Raft due to its adoption in production systems [45, 46, 47, 48, 49], and the availability of standalone production-quality implementations [50].

6.1. Storage System Implementation

We have implemented FlairKV, including all switch data plane features, the FLAIR module, leaders' and followers' modifications, and the client library. We extended the Raft's follower code to implement an advance_then_read() function. We extended the leader to notify the *lflair* module as soon as it gets elected, and to extract its commit_index, the list of followers with a commit_index equal to the leader's commit_index, and the list of uncommitted writes. We extended the write reply with the list of followers which acknowledged the write. We implemented the leader lease optimization [8, 24] and modified Raft's client library to add the FLAIR header to client requests.

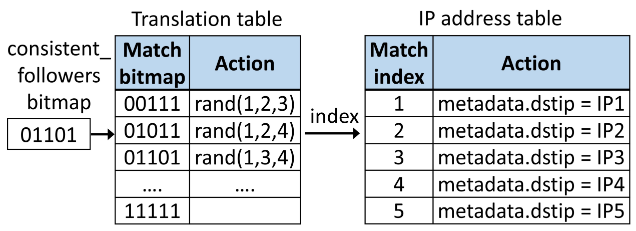
6.2. Switch Data Plane Implementation

The switch data plane is written in P4 v14 [31] and is compiled for Barefoot's Tofino ASIC [33], with Barefoot's P4Studio software suite [51]. Our P4 code defines 30 tables and 12 registers: six for the session array and six for the kgroup array. The kgroup array has 4K entries. Larger number of kgroups had negligible effect on performance. In total, our implementation uses less than 5% of the on-chip memory available in the Tofino ASIC, leaving ample resources to support other switch functionalities or more FlairKV instances. The rest of this section discusses optimizations implemented in FlairKV to cope with the strict timing and memory constraints of P4 and switch ASIC.

Heartbeats implementation. The leader and the switch exchange periodic heartbeats. If the switch_stepdown period passes without receiving a leader heartbeat, the switch deactivates the session. Instead of running a process in the controller to continuously track heartbeats, the switch monitors missed heartbeats as part of the validation step in the processing pipeline. The switch keeps track of the timestamp of the last heartbeat received in the session array (Listing 1). When processing any FLAIR packet, the switch computes the difference between the current time and the last heartbeat timestamp; if the difference is larger than switch_stepdown, the switch deactivates the session, making the system unavailable until the leader starts a new session.

Forwarding logic translates the consistent followers' bitmap to follower IP addresses. Storing the IP addresses of consistent followers for every entry in the kgroup array significantly increases the memory footprint. Moreover, randomly selecting a follower from the list while avoiding inconsistent ones is tricky given the P4 and current ASIC challenges (Section 2.2). Instead, the FlairKV leader encodes the follower status in a one-byte `consistent_followers` bitmap (Listing 1). Replicas are ordered in a list. If the least significant bit in the `consistent_follower` bitmap is set, this indicates that the first replica in the list is consistent, and so forth.

When forwarding a read request, the switch translates the encoded bitmap of consistent followers to select one follower; Figure 6 shows the translation process. The `consistent_followers` bitmap is used as an index to the translation table. Each entry in the table has an action that randomly selects a number that is then used as an index to the IP addresses table.



Match bitmap	Action
00111	rand(1,2,3)
01011	rand(1,2,4)
01101	rand(1,3,4)
....	....
11111	

Match index	Action
1	metadata.dstip = IP1
2	metadata.dstip = IP2
3	metadata.dstip = IP3
4	metadata.dstip = IP4
5	metadata.dstip = IP5

Figure 6. Logical view of the forwarding logic. The stability bitmap matches an entry in the translation table and executes the corresponding action, generating an index of the selected destination's IP address. Using the index, the IP address table sets the destination's IP address in the metadata.

This design has two benefits: it significantly reduces the memory footprint of the kgroup array, and it can be accelerated using P4 "action profiles" [52].

Load balancing. In addition to the aforementioned random load-balancing technique (Figure 6), we implemented two load-aware techniques:

- *Leader avoidance*. Our benchmarking revealed that the write operation takes 35 times longer than a read operation; most of this overhead is borne by the leader. Consequently, this load-balancing technique avoids sending read requests to the leader for stable kgroups if there are any writes in the system. The aim is to reduce the leader load, as it is already busy serving writes and serving reads for unstable kgroups.
  To implement this technique, we compare the sequence number of a write_reply with the `session_seq_num`. If they are not equal, then there are pending writes in the system and the leader should not be burdened with any reads to stable kgroups.

- *Follower load awareness*. This technique distributes the load across followers proportionally to their load in the last *n* seconds. This technique is especially useful for deployments that use heterogeneous hardware, experience workload variations, or deploy more than one replica (i.e., replicas for different ranges of keys) on the same machine. In our design, followers report the length of the request queue in every heartbeat. Every second, the leader calculates the average queue length for each follower and assigns proportional weights to each follower. The leader updates the translation table to reflect these weights. For instance, if follower 1 should receive double the load of any other replica, the action for a bitmap 00111 will be `rand`(1, 1, 2, 3), doubling the chance replica 1 is selected.

Register access logic. Each stage has its own dedicated registers, and a register can be accessed only once in a stage. This restriction complicates FlairKV's logic, as different packet types (e.g., read and write_reply) must access the same registers at different stages in the pipeline. To cope with this restriction, FlairKV adds a dedicated table to access each register. Figure 7 shows an example of an action table for accessing register *r1*. Our code aggregates the information about all possible modes of accessing *r1* in the packet's metadata, including the access type (read or write), the index, and which data should be written or where the value should be read to. We then use a dedicated match-action table (Figure 7) to perform the actual read or write operation to/from the register in a single stage with a single invocation of the table. This approach has the additional benefit of reducing the number of stages.

Processing concurrent requests. The switch processes packets sequentially in a pipeline. Each pipeline stage processes one packet at a time. The switch may have multiple

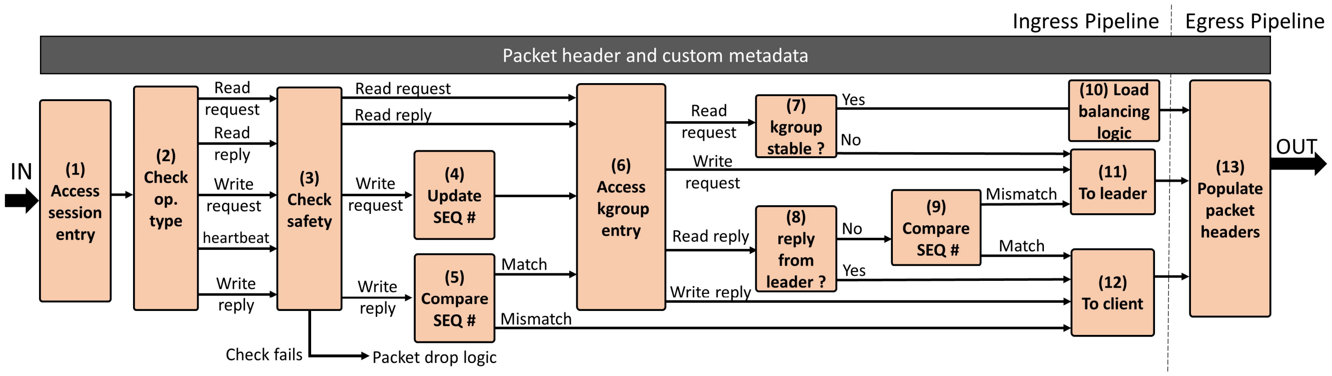


Figure 8. Logical view of the FlairKV switch data plane.

pipelines, each serving a subset of switch ports. FLAIR uses a single ingress pipeline and all egress pipelines. If a FLAIR packet is received on a different ingress pipeline, the packet is recirculated [52] to the FLAIR pipeline.

6.3. Putting the Switch Pipeline Together

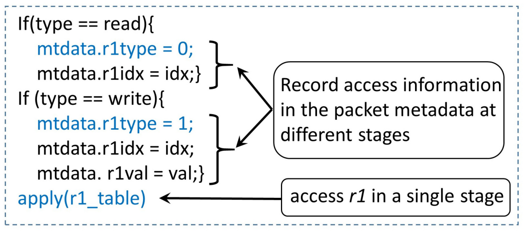
Figure 8 shows the pipeline layout in the switch data plane and the flow for a FlairKV packet. The pipeline starts by reading the session information (1 in Figure 8) and adding it to the packet metadata. Then the it extracts the operation type (2) and validates the request (3) by verifying the session id. If the packet has an older session id the packet is dropped. Further, in the validation stage the switch confirms that it did not miss leader heartbeats in the last switch_stepdown period (Section 4.6), else it deactivates the session.

Read requests access the kgroup array (6), and if the group is stable, the request is forwarded to a load-balancing logic (10) that implements the forwarding logic (Section 6.2); otherwise, it is sent to the leader.

If a read reply is from the leader, it is forwarded to the client (12). If it is from a follower, the pipeline performs the safety check (9) and, if it suspects the reply is stale, drops the reply, then resubmits the read request to the leader (11).

Write requests update the session_seq_num (4) and the kgroup entry (6), then are sent to the leader (11).

Write replies compare the sequence number of the reply to the one in the kgroup entry (5); if they match, the kgroup


```
If(type == read){
   mtdata.r1type = 0;
   mtdata.r1idx = idx;}
If (type == write){
   mtdata.r1type = 1;
   mtdata.r1idx = idx;
   mtdata. r1val = val;}
apply(r1_table)
```
Record access information in the packet metadata at different stages
access *r1* in a single stage

Match	Action
mtdata.r1type == 0	mtdata.r1val = reg1[mtdata.r1idx]
mtdata.r1type == 1	reg1[mtdata.r1idx] = mtdata.r1val

Register r1 access table (**r1_table**)

Figure 7. Register access table. P4 code aggregates access information that is used by a dedicated register access table.

entry is updated (6) and the pipeline forwards the reply to the client (12).

The egress pipeline (13) has one logical stage that populates the header fields (e.g., SEQ number, SID, etc.) using the data available in the packet's metadata.

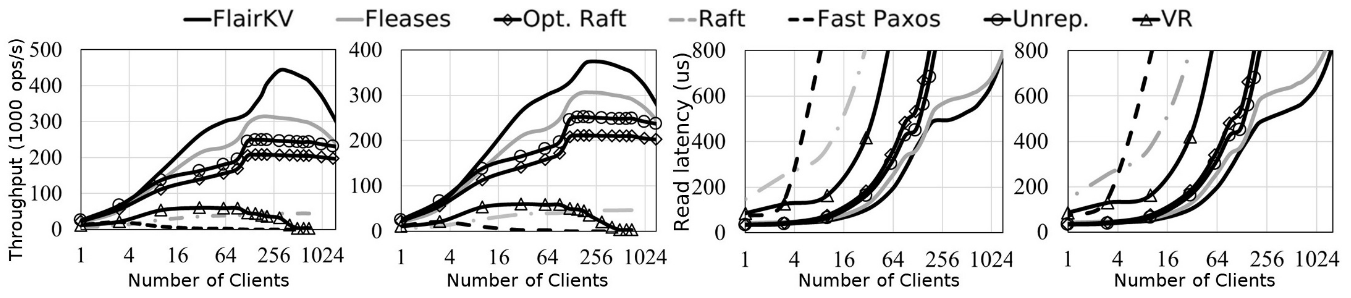
7. Evaluation

We compare our prototype with previous approaches in terms of throughput and latency (§7.1) with different workload skewness (§7.2) and read/write ratios (§7.3).

Testbed. We conducted our experiments using a 13-node cluster. Each node has an Intel Xeon Silver 10-core CPU, 48GB of RAM, and 100Gbps Mellanox NIC. The nodes are connected to an Edgecore Wedge 100 ×32BF switch with 32 100Gbps ports. The switch has Barefoot's Tofino ASIC, which is P4 programmable. Unless otherwise specified, three machines ran the server code, while the other 10 machines generated the workload.

Alternatives. We compare the throughput and latency of the following designs and optimizations:

- **Leader-based**. We used two leader-based protocol implementations: LogCabin, the original implementation of Raft (**Raft**), and an implementation of Viewstamped replication (**VR**) [26]. Raft and VR implement a batching optimization which batches and replicates multiple log entries in a single round trip.
- **Optimized Leader-based (Opt. Raft).** Our benchmarking revealed that the original Raft implementation could not utilize the resources of our cluster. We implemented two main optimizations: first, we changed the request-processing logic from an event-driven to a thread-pool design, as our benchmarking indicated a thread-pool performs better; second, we implemented the leader-lease optimization. These changes significantly improved Raft's performance.
- **Quorum-based reads (Fast Paxos).** An alternative to the leader-based design is the quorum design [40, 41, 53]. Typically, client read requests are sent to all followers, and each follower responds directly to the client. The client waits for a reply from a supermajority [53] before



(a) Throughput - Uniform (b) Throughput -Zipf (c) Latency-Uniform (d) Latency-Zipf

Figure 9. Throuhgput and Latency while varying the number of clients. The figures show the throughput and the average latency for different number of clients for workload B for the uniform distribution (a, c), and for the Zipf distribution (b, d).

completing a read. We used a Fast Paxos implementation that implements only the normal case [26].

- **Follower-lease optimization (FLeases).** Similar to MegaStore [1], the leader grants read leases to all followers. Before serving a write, the leader revokes all leases, processes the write operation, and then grants a new lease to followers. The lease's grant/revoke messages are piggybacked on the consensus protocol messages. However, writes should be processed by *all* followers before replying to the client. In our experiments, if a follower receives a read request for an object for which it does not have an active lease, it forwards the request to the leader. MegaStore applications typically partition the keys into thousands of groups, each group contains logically-related keys [1] (e.g., a key group per blog [1]). We partitioned the keys into 4K groups (the same number of kgroups in FlairKV), and followers get a lease per group. Clients randomly select a follower for each read request and send the request directly to it.
- **Unreplicated/NOPaxos (Unrep.).** As a baseline, the unreplicated configuration deploys Optimized-Raft (discussed above) on a single node. The single node stores the data set and serves all operations without replication.

  This configuration also represents the best possible performance of the network-optimized NOPaxos [40] protocol. NOPaxos uses a network switch to order and multicast read and write operations to all replicas. An operation is successful if the majority accepts a write or returns the same value for a read. Consequently, NOPaxos read performance is limited by the slowest node in the majority of nodes. NOPaxos evaluation shows that the best throughput and latency the protocol can achieve are within 4% that of an unreplicated system [40].
- **FlairKV.** Unless otherwise specified, we used FlairKV with the leader-avoidance load-balancing technique.

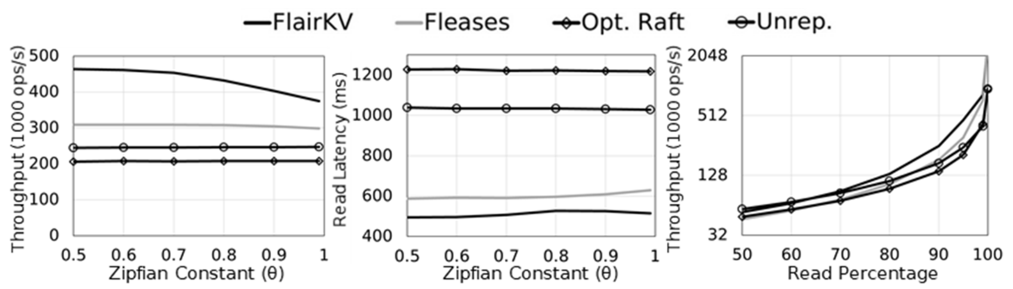
We benchmarked every system and selected a configuration that maximized its performance. We stored all data in memory. In all experiments, all systems' performance (with the exception of FastPaxos) was stable with a standard deviation less than 1%.

Workload. We used synthetic benchmarks and the YCSB benchmark [54] to evaluate the performance of all systems. In our evaluation, we considered both uniform and skewed workloads. The skewed workload follows the Zipf distribution with a skewness parameter of 0.99. We also used the YCSB benchmark. We experimented with 100,000 and 1 million keys. We present the results with 100,000 keys as, in skewed workloads, the fewer number of hot keys increased the chance of having concurrent requests accessing the same key (i.e. is less favorable for FlairKV). FlairKV brings slightly higher performance benefit when using 1 million keys than 100,000 keys. The key size is 24 bytes and the hash of the key string is used as the key in the FLAIR protocol. The value size is 1KB.

7.1. Performance Evaluation

We compared the seven systems using YCSB workload B (95:5 read:write ratio) while varying the number of clients, with uniform and skewed workload distribution. Figure 9 shows the throughput and average latency with a uniform and skewed distributions. With the uniform distribution (Figure 9 (a) and (c)), FlairKV achieves up to 42% higher throughput and 23.7% lower average latency than FLeases, and 1.3 to 2.1 times higher throughput and 1.5 to 2.4 times lower latency compared to optimized Raft and unreplicated setup. Fast Paxos, Raft, and VR, achieve the lowest throughput and highest latency as these systems contact the majority of nodes for every read.

FlairKV achieved better performance than FLeases for three reasons. First, FlairKV uses the leader-avoidance load-balancing technique, which reduces the load on the leader when there are writes, thereby accelerating writes and shortening the time period in which kgroups are marked unstable. This approach is effective as writes take almost 35 times longer than reads in Opt.Raft, and 30 times longer in the unreplicated setup. We recorded the number of read requests served by the leader. For instance, with 300 clients (Figure 9.a) the leader served 2% of the reads in FlairKV (those are reads to unstable kgroups), while it served 34% of the reads in FLeases. We note that the leader-avoidance technique cannot be applied to FLeases which tasks the clients



(a) Throughput (b) Read latency

Figure 10. Throughput and Latency while varying skewness. The figures show the throughput (a) and the average latency (b) for different zifpian constants for a uniform workload B with 300 clients.

Figure 11. Throughput while varying the read ratio. Using uniform workload B.

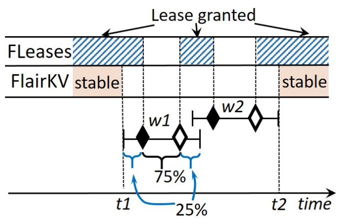


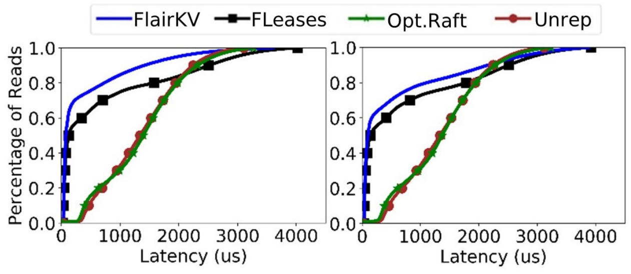
Figure 12. Subtle effect of FLAIR. FLeases may grant leases for up to 25% more time compared to FlairKV. Bars mark the time from the moment a switch receives a write request (w1 or w2) until it receives a corresponding reply.

with selecting a follower to send the read request to. This technique requires accurate information about the current load of the leader and which followers are stable which are not available to clients.

Second, in FLeases, when an object is not stable, if a client sends a request to a follower, the follower will redirect the request to the leader, increasing overhead and incurring extra latency. Unlike FLeases, FlairKV switch knows if an object is not stable and forwards read requests for that object directly to the leader. The third reason which had a minor impact when using 3 replicas is that the write operation in FLeases need to reach all followers, while FlairKV writes only need a majority.

Optimized-Raft's performance is better than that of Raft, VR, and FastPaxos. The unreplicated deployment slightly improves throughput and latency over Optimized-Raft by avoiding the replication overhead for write operations. These two systems still lag behind FlairKV as they only utilize a single node (the leader) for serving all reads and writes.

Figure 9 (b) and (d) show the throughput and average latency with a skewed workload (Zifpian constant of 0.99). The skewed workload results in higher contention and an increased frequency at which a read request finds a kgroup unstable. This contention reduces the chances of reading from followers. FlairKV leader served 21% of reads of which 1% are redirected from followers, while FLeases leader served 37% of reads. Even under the skewed workload, FlairKV still achieves the highest performance, up to 26% higher throughput and 18.1% lower latency than FLeases, and 1.5 to 1.8 times higher throughput and 2 to 2.4 times lower latency than optimized Raft and the unreplicated setup.



(a) B-Uniform (b) B-Zipf

Figure 13. Latency CDF. The figures show the latency CDF for reads under workload B using 300 clients with a uniform distribution (a), and a Zipf distribution with skewness of 0.99 (b). The lines for Opt. Raft and Unrep. almost overlap.

Latency evaluation. Figure 13.a shows the latency CDF of FlairKV, FLeases, OptRaft, and Raft. Under the uniform workload B with 300 clients (other workloads had similar results). FlairKV lowered the latency for the slowest 40% requests by at least 38% relative to FLeases. Under the Zipf workload (Figure 13.b), FlairKV lowered the slowest 50% of request by up to 35% relative to FLeases.

FLeases has higher latency as it incurs extra delay due to the load imbalance between nodes (e.g., the leader serves 41% of requests for workload B with Zipf distribution) and due to followers redirecting 4% of requests to the leader.

Under all workloads, FlairKV significantly improved operation's latency relative to OptRaft and Raft. The median latency of FlairKV is 2% of Raft's latency and 2-8% of OptRaft's latency.

7.2. Workload Skewness

We measured the impact of the workload skewness on throughput (Figure 10.a) and average latency (Figure 10.b) by varying the Zipfian constant from 0.5 to 0.99. FlairKV consistently achieves better performance: 1.26 to 2.25 times higher throughput and 1.13 to 2.48 times lower average latency compared to all other systems. We notice that as the skewness increases FlairKV and FLeases performance decreases as higher skewness increases contention on the few popular kgroups, making them unstable for longer time, and increases the number of requests the leaders have to process. Other systems performance is not noticeably affected by skewness.

We noticed high workload skewness affects FlairKV's performance more than FLeases. This is due to a subtle side effect of FlairKV. When there are concurrent writes to the same kgroup, FlairKV will mark a group unstable from the moment the first request is processed by the switch until the last request to the kgroup is replied to ([*t1*, *t2*] in Figure 12). In FLeases, the lease revocation is piggybacked on the write replication step (black diamonds in Figure 12). Once the leader commits a write, it sends a commit notification and grants a new lease to the followers (white diamonds). Hence, FLeases may grant a lease between concurrent writes, creating more opportunity for serving reads from followers.

To further understand this effect, we tracked leases and the stability of kgroups under the skewed (factor of 0.99) write heavy YCSB workload A (1:1 read:write ratio). We noticed that while 29% of reads found the kgroup unstable in FlairKV, only 4% of reads in FLeases reached a follower that did not have a lease. We further profiled the write operation path and found that FLeases revokes leases for 75% of the write operation time (Figure 12), 25% shorter than the period FlairKV marks a kgroup unstable. Despite this subtle effect FlairKV leader still has lighter load, it served 29% of reads compared to 37% served by the FLeases leader. Notwithstanding this effect FlairKV still brings 17% to 26% performance improvement even under skewed workloads.

7.3. Read/Write Ratio

Figure 11 shows the effect of the ratio of reads to writes on systems' performance with a uniform workload B. Compared to FLeases, FlairKV has up to 1.5 times higher throughput for all read to write ratios, with the exception of the read-only workload in which their performance is comparable. FlairKV has 1.25 to 2.8 times higher throughput compared to the Opt. Raft. Compared to the unreplicated setup, FlairKV has up to 2.8 times higher throughput for workloads with 70% reads or more and a comparable performance under write heavy workloads (read ratio 50-70%).

8. Related Work

Network-accelerated systems. Recent projects have utilized SDN capabilities to provide load balancing [55, 56, 57], access control [58], seamless virtual machine migration [59], and improving system security, virtualization, and network efficiency [60]. SwitchKV [29] uses SDN capabilities to route client requests to the caching node serving the key. A central controller populates the forwarding rules to invalidate routes for objects that are being modified and installs routes for newly cached objects. NetCache [28] proposes using the limited switch memory as a look-through cache for key-value stores.

Network-accelerated consensus. A number of recent efforts leverage SDN's capabilities to optimize consensus protocols. Speculative Paxos [41] builds a mostly ordered multicast primitive and uses it to optimize the multi-Paxos consensus protocol. Network-ordered Paxos (NOPaxos) [40] leverages modern network capabilities to order multicast messages and add a unique sequence number to every client request. NOPaxos uses these sequence number to serialize operations and to detect packet loss. Speculative Paxos and NOPaxos are optimized for operations that update the log but not for read operations. NetChain [61] and NetPaxos [62] implement replication protocols on a group of switches. These protocols are suitable for systems that store only a few megabytes of data (e.g., 8MB in the NetChain prototype). Unlike FLAIR, these efforts do not optimize for read operations. Reads are still served by the leader or a quorum of replicas.

Consensus protocols optimized for the WAN. A number of consensus protocols are optimized for WAN deployments. Quorum leases [14] proposes giving a read lease to some of the followers; Unlike Megastore leases, when an object is modified, only the followers that have the lease are contacted. Quorum leases has a better performance than Megastore leases in WAN setups, but do not bring benefits when deployed in a single cluster [14]. Mencius [63] is a multi-leader protocol in which each leader controls part of the log. EPaxos [64] is a leaderless protocol where clients can submit request to any replica. Non-conflicting write can commit in one round trip, while conflicting writes will be resolved using Paxos.

CURP [65] optimizes the write operation through exploiting commutativity between concurrent writes. In data center deployments, CURP reads are served by the leader and hence are limited to a single node performance, in WAN deployment CURP applies a technique similar to FLeases.

9. Conclusion

We present FLAIR, a novel protocol that leverages the capabilities of the new generation of programmable switches to accelerate read operations without affecting writes or using leases. FLAIR identifies, at line rate, which replicas can serve a read request consistently, and implements a set of load-balancing techniques to distribute the load across consistent replicas. We detailed our experience building FlairKV and presented a number of techniques to cope with the restrictions of the current programmable switches. We hope our experience informs a new generation of systems that co-design network protocols with system operations.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Amin Vahdat, for their insightful feedback. We thank Ali Mashtizadeh, Bernard Wong, and Ken Salem for their insightful feedback on early versions of this paper. We thank Lori Paniak for his help in running the experiments. This research was supported by an NSERC Discovery grant, Canada Foundation for Innovation (CFI) grant, and a grant from Waterloo-Huawei Joint Innovation Lab.

References

[1] J. Baker, C. Bond, J. C. Corbett *et al.*, "Megastore: Providing scalable, highly available storage for

interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011.

[2] P. Hunt, M. Konar, F. P. Junqueira *et al.*, "ZooKeeper: wait-free coordination for internet-scale systems," in *Proceedings of the USENIX annual technical conference*, Boston, MA, 2010.

[3] B. Calder, J. Wang, A. Ogus *et al.*, "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011, doi: 10.1145/2043556.2043571.

[4] J. C. Corbett, J. Dean, M. Epstein *et al.*, "Spanner: Google's globally-distributed database," in *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, USA, 2012: USENIX Association.

[5] N. Bronson, Z. Amsden, G. Cabrera *et al.*, "TAO: Facebook's distributed data store for the social graph," in *Proceedings of the USENIX Technical Conference*, San Jose, CA, 2013: USENIX Association.

[6] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., 2004.

[7] L. Lamport, "Paxos made simple," *ACM Sigact News,* vol. 32, no. 4, pp. 18-25, 2001.

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the USENIX Annual Technical Conference*, Philadelphia, PA, 2014: USENIX Association.

[9] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems&Networks*, 2011: IEEE Computer Society, doi: 10.1109/dsn.2011.5958223.

[10] B. Liskov and J. Cowling, "Viewstamped replication revisited," Technical Report MIT-CSAIL-TR-2012-021, MIT, 2012.

[11] J. Shute, R. Vingralek, B. Samwel *et al.*, "F1: a distributed SQL database that scales," *Proc. VLDB Endow.,* vol. 6, no. 11, pp. 1068-1079, 2013, doi: 10.14778/2536222.2536232.

[12] B. Atikoglu, Y. Xu, E. Frachtenberg *et al.*, "Workload analysis of a large-scale key-value store," presented at the Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems, London, England, UK, 2012.

[13] C. Gray and D. Cheriton, "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1989: ACM, doi: 10.1145/74850.74870.

[14] I. Moraru, D. G. Andersen, and M. Kaminsky, "Paxos Quorum Leases: Fast Reads Without Sacrificing Writes," in *Proceedings of the ACM Symposium on Cloud Computing*, Seattle, WA, USA, 2014: ACM, doi: 10.1145/2670979.2671001.

[15] J. Terrace and M. J. Freedman, "Object storage on CRAQ: high-throughput chain replication for read-mostly workloads," presented at the Proceedings of the 2009 conference on USENIX Annual technical conference, San Diego, California, 2009.

[16] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, Washington, 2006: USENIX Association.

[17] "Swift's documentation." https://docs.openstack.org/swift/stein/index.html (accessed April 14, 2019.

[18] "Redis." https://redis.io (accessed April 14, 2019.

[19] "Apache Cassandra." https://cassandra.apache.org (accessed April 14, 2019.

[20] G. DeCandia, D. Hastorun, M. Jampani *et al.*, "Dynamo: amazon's highly available key-value store," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, Stevenson, Washington, USA, 2007: ACM, doi: 10.1145/1294261.1294281.

[21] D. B. Terry, V. Prabhakaran, R. Kotla *et al.*, "Consistency-based service level agreements for cloud storage," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Farminton, Pennsylvania, 2013: ACM, doi: 10.1145/2517349.2522731.

[22] B. F. Cooper, R. Ramakrishnan, U. Srivastava *et al.*, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.,* vol. 1, no. 2, pp. 1277-1288, 2008, doi: 10.14778/1454159.1454167.

[23] M. Poke and T. Hoefler, "DARE: High-Performance State Machine Replication on RDMA Networks," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, Portland, Oregon, USA, 2015, 2749267: ACM, pp. 107-118, doi: 10.1145/2749246.2749267.

[24] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the annual ACM symposium on Principles of distributed computing*, Portland, Oregon, USA, 2007: ACM, doi: 10.1145/1281100.1281103.

[25] D. Mazieres, "Paxos made practical," ed, 2007.

[26] "NOPaxos consensus protocol." https://github.com/UWSysLab/NOPaxos (accessed April 14, 2019.

[27] S. Al-Kiswany, S. Yang, A. C. Arpaci-Dusseau *et al.*, "NICE: Network-Integrated Cluster-Efficient

Storage," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, Washington, DC, USA, 2017: ACM, doi: 10.1145/3078597.3078612.

[28] X. Jin, X. Li, H. Zhang *et al.*, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, 2017: ACM, doi: 10.1145/3132747.3132764.

[29] X. Li, R. Sethi, M. Kaminsky *et al.*, "Be fast, cheap and in control with SwitchKV," in *Proceedings of the Usenix Conference on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, 2016: USENIX Association.

[30] "LogCabin storage system." https://logcabin.github.io (accessed April 14, 2019.

[31] "P4." https://p4.org (accessed April 14, 2019.

[32] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.,* vol. 16, no. 2, pp. 133-169, 1998, doi: 10.1145/279227.279229.

[33] "Barefoot Tofino." https://www.barefootnetworks.com/products/brief-tofino/ (accessed April 14, 2019.

[34] "Cavium / XPliant." https://origin-www.marvell.com/documents/netpxrx94dcdhk8sksbp/ (accessed April 14, 2019.

[35] "High-Capacity StrataXGS® Trident 3 Ethernet Switch Series." https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series (accessed September 9, 2019.

[36] P. Bosshart, D. Daly, G. Gibb *et al.*, "P4: programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.,* vol. 44, no. 3, pp. 87-95, 2014, doi: 10.1145/2656877.2656890.

[37] "Data Center: Load Balancing Data Center." https://learningnetwork.cisco.com/docs/DOC-3438 (accessed April 14, 2019.

[38] L. A. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009, p. 120.

[39] N. McKeown, T. Anderson, H. Balakrishnan *et al.*, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.,* vol. 38, no. 2, pp. 69-74, 2008, doi: 10.1145/1355734.1355746.

[40] J. Li, E. Michael, N. K. Sharma *et al.*, "Just say no to paxos overhead: replacing consensus with network ordering," in *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, USA, 2016: USENIX Association.

[41] D. R. K. Ports, J. Li, V. Liu *et al.*, "Designing distributed systems using approximate synchrony in data center networks," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Oakland, CA, 2015: USENIX Association.

[42] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, Samer Al-Kiswany, "Accelerating Reads with In-Network Consistency-Aware Load Balancing," Technical Report, Waterloo Advanced Systems Lab, 2020.

[43] "TLA+ Language." https://lamport.azurewebsites.net/tla/tla.html (accessed April 14, 2019.

[44] D. Ongaro. "Raft TLA+ Specification." https://github.com/ongardie/raft.tla (accessed 2019).

[45] "etcd: Distributed reliable key-value store for the most critical data of a distributed system." https://github.com/etcd-io/etcd (accessed April 14, 2019.

[46] "RethinkDB: the open-source database for the realtime web." https://www.rethinkdb.com/ (accessed April 14, 2019.

[47] "Open Network Operating System (ONOS) - Cluster Coordination." https://wiki.onosproject.org/display/ONOS/Cluster+Coordination (accessed.

[48] "Apache Kudu - Fast Analytics on Fast Data." https://kudu.apache.org/ (accessed April 14, 2019.

[49] "Hashicorp Raft implementation." https://github.com/hashicorp/raft (accessed April 14, 2019.

[50] "The Raft Consensus Algorithm." https://raft.github.io/ (accessed April 14, 2019.

[51] "Barefoot P4 Studio." https://www.barefootnetworks.com/products/brief-p4-studio/ (accessed April 14, 2019.

[52] "P4 v16 Portable Switch Architecture (PSA)." https://p4.org/p4-spec/docs/PSA-v1.0.0.html (accessed April 14, 2019.

[53] L. Lamport, "Fast paxos," *Distributed Computing,* vol. 19, no. 2, pp. 79-103, 2006.

[54] "Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB." https://github.com/basicthinker/YCSB-C (accessed April 14, 2019.

[55] B. Cully, J. Wires, D. Meyer *et al.*, "Strata: High-performance scalable storage on virtualized non-volatile memory," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2014, pp. 17-31.

[56] N. Handigol, M. Flajslik, S. Seetharaman *et al.*, "Aster* x: Load-balancing as a network primitive," in *GENI Engineering Conference (Plenary)*, 2010, pp. 1-2.

[57] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based server load balancing gone wild," in *Proceedings of the USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Boston, MA, 2011: USENIX Association.

[58] A. K. Nayak, A. Reimers, N. Feamster *et al.*, "Resonance: dynamic access control for enterprise networks," in *Proceedings of the ACM workshop on Research on enterprise networking*, Barcelona, Spain, 2009: ACM, doi: 10.1145/1592681.1592684.

[59] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin *et al.*, "XvMotion: unified virtual machine migration over long distance," in *Proceedings of the USENIX Annual Technical Conference*, Philadelphia, PA, 2014: USENIX Association.

[60] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE communications surveys & tutorials,* vol. 16, no. 1, pp. 493-512, 2014.

[61] X. Jin, X. Li, H. Zhang *et al.*, "Netchain: scale-free sub-RTT coordination," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Renton, WA, USA, 2018: USENIX Association.

[62] H. T. Dang, D. Sciascia, M. Canini *et al.*, "NetPaxos: consensus at network speed," in *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research*, Santa Clara, California, 2015: ACM, doi: 10.1145/2774993.2774999.

[63] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for WANs," presented at the Proceedings of the 8th USENIX conference on Operating systems design and implementation, San Diego, California, 2008.

[64] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in Egalitarian parliaments," presented at the Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, Farminton, Pennsylvania, 2013.

[65] S. J. Park and J. Ousterhout, "Exploiting commutativity for practical fast replication," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 47-64.

Towards Logically Centralized Interdomain Routing

Shahrooz Pouryousef, Lixin Gao, and Arun Venkataramani

University of Massachusetts Amherst

Abstract

In this paper, we present the design and implementation of CIRCA, a logically centralized architecture and system for interdomain routing that enables operators to offload BGP-style route computation to the cloud while preserving the confidentiality of proprietary information. To this end, our work presents the first provably safe, live, and fully distributed convergence detection algorithm for decentralized policy routing and, somewhat surprisingly, shows that long MRAI timers can likely be completely eliminated while significantly improving convergence delays with logical centralization. Our experiments with a Quagga-based CIRCA prototype and the Internet's AS topologies suggest that CIRCA can improve interdomain routing convergence delays and transient route inconsistencies by over an order of magnitude and offers non-trivial incremental deployability benefits with modest changes to widely deployed routing infrastructure.

1 Introduction

Logical centralization of control and management for enterprise networks has proven successful in recent years. However, this trend has minimally, if at all, affected interdomain routing in the Internet that has seen little fundamental change in over two decades of operation and continues to suffer from long convergence delays, lack of control-data separation, poor management knobs, lack of evolvability, etc.

Logical centralization or cloud-assisted interdomain route computation holds the promise of alleviating these longstanding problems but is not easy to accomplish. Unlike enterprise networks, a key stumbling block is the need to maintain the confidentiality of proprietary routing policies. Recent research [2, 17] has attempted to attack this problem by employing secure multiparty computation, but its inherently computationally-expensive nature poses a scaling challenge, so it has been demonstrated only at small scales with restrictive assumptions on the expressiveness of routing policies. Cloud-assisted interdomain route computation has therefore

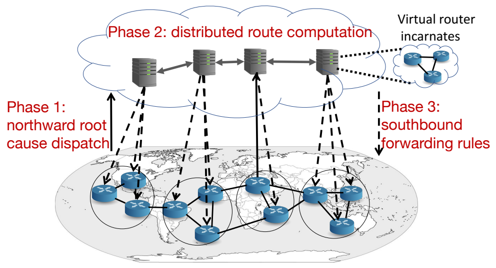

Phase 2: distributed route computation
Virtual router incarnates
Phase 1: northward root cause dispatch
Phase 3: southbound forwarding rules

Figure 1: CIRCA: northward root cause dispatch, cloud-driven route computation, and southbound forwarding rules.

been limited to narrower contexts such as software-defined exchange points [15, 16], engineering traffic across multiple ASes owned by a single enterprise [18, 53], or compromising on either or both of scale and policy confidentiality [26].

In this paper, we present the design and implementation of CIRCA, the first logically centralized interdomain routing control architecture and system enabling operators to offload BGP-style route computation *as-is* to the cloud while preserving the confidentiality of proprietary policies. The CIRCA service makes the following assurances to network operators: (1) forwarding entries computed by CIRCA will be *equivalent* (as formalized in §3.4.1) to what their routers are computing today; (2) CIRCA will quickly return forwarding entries reflecting a converged state of the network circumventing BGP's long tail of convergence delays (tens of seconds to minutes); and (3) CIRCA does not require an AS to disclose proprietary policy information to any third party except to the extent that it can already be inferred by its eBGP peers.

The high-level design of the baseline CIRCA system, as illustrated in Figure 1, is simple: each router speaking BGP (or *ground router*) has a virtual router incarnate (or *avatar*) in the CIRCA cloud. Upon detecting a root cause event, a ground router dispatches it cloudward to its avatar; the virtual routers in the cloud thereafter simply replay the same BGP protocol as ground routers but in a significantly "fast-forwarded" manner; and upon convergence, cloud routers dispatch modified forwarding entries to their ground incarnates.

A natural question is why this simple, perhaps even seemingly naive, high-level design can enable the cloud to compute routes faster than the ground if it's replaying the same protocol. The answer is threefold: (1) the cloud can easily afford orders of magnitude more control bandwidth, e.g., tens or even hundreds of Gbps of LAN bandwidth, compared to the ground; (2) propagation delays in the cloud ($<$ 1 ms or even just a few microseconds [21, 33, 35, 55]) are several orders of magnitude lower than the delay diameter of the Internet ($\approx$ hundreds of ms); (3) the (cloud) control plane is physically isolated from and can not interfere with the (ground) data plane. Thus, the cloud has the luxury of doing away with long route advertisement timers that in ground-BGP today are believed necessary to mitigate the risk of super-exponential message complexity [9, 28, 29, 36, 37] and conservatively set to high values (e.g., 30s is a common vendor default).

The deceptively simple exposition above hides a number of research questions that must be answered to translate the high-level design to a deployable system in practice. Can the cloud really compute stable routing outcomes an order of magnitude faster than the ground? Can (and how?) cloud routers quickly detect that the distributed processing of a root event has converged? Can CIRCA guarantee consistency of computed routes and ensure that its computed routing outcomes will match those of BGP despite component or network failures? Can CIRCA coexist with BGP and is it incrementally deployable for incremental benefit?

Tackling the above questions and answering them in the affirmative aided by an implemented prototype of CIRCA is our primary contribution comprising the following parts:

1. *Distributed convergence detection*: Design and implementation of the first fully distributed BGP convergence detector that is provably safe, i.e., no false positives, and live, i.e., detection incurs a modest bounded delay (§3.3).

2. *Quick end-to-end convergence*: Large-scale prototype-driven experiments showing that CIRCA can ensure predictably quick convergence reducing BGP's tail convergence delays by over an order of magnitude, in part by eliminating unnecessary long timers (§3.2, §4.1).

3. *Route computation equivalence*: A design that provably ensures that its computed routing outcomes are equivalent to those of any *convergent* (formalized in §3.4) distributed policy-based ground routing protocol.

4. *Incremental deployability*: Design and evaluation of mechanisms to deploy CIRCA *co-existent* (§3.5.1) with BGP as well as in an *incremental* manner (§3.5.2).

2 Background and lineage

Logical centralization of network control and management, including for interdomain routing, has a long scientific lineage. We overview what prior work has accomplished on that front to position how CIRCA builds upon that work.

In intradomain routing, a line of work seemingly starting with calls for "routing as a service" [30] or "separating routing from routers" followed by works such as RCP [4], 4D [52], Ethane [5] etc. spurred what is now known as software-defined networking (SDN), a term that commonly means logical centralization of control and management for enterprise networks. Much follow-on SDN research as well as widespread embrace by industry firmly attests to its benefits such as cleaner control and data separation, ease of management with a global view, hardware-software decoupling, etc.

Interdomain routing on the other hand, since BGP's creation in the late 80s and progressive standardization through the 90s, has remained largely untouched by the logical centralization trend as BGP4 continues to be the de facto interdomain routing protocol. Recent research however has begun to explore extending SDN approaches to the interdomain context [1, 41, 44, 54]. For example, SDX by Gupta et al. [16] is a software-defined Internet exchange point (IXP) that enables more sophisticated packet processing rules to engineer interdomain traffic. Kotronis et al. [10, 24–27] also advocate a logically centralized “multi-domain SDN" approach as an alternative to BGP, introduce a publicly available (single-node) Mininet-based emulation platform for experimenting with hybrid BGP-SDN routing, and show modest improvements in convergence delays because of centralization. Gupta et al. [2, 17] advocate using secure multiparty computation (SMPC) so as to combine the benefits of logical centralization with confidentiality of proprietary routing policies, and argue that SMPC may be computationally feasible for BGP with some policy restrictions. Recent research has developed approaches for preserving the privacy of ISP policies at Internet exchange points [6, 7, 14].

Consensus routing [20] advocates a consistency-first (in contrast to the Internet's longstanding allegiance to soft-state) approach to interdomain routing. It works by periodically capturing a consistent snapshot of global network state—e.g., by having a small number of say tier-1 ASes engage in a consensus protocol to agree upon globally in-propagation updates—so as to enable computation of so-called *stable forwarding tables* that are guaranteed to be loop-free while relying on *transient* routing heuristics in the forwarding plane to re-route packets encountering blackholes. Consensus routing can be viewed as logically centralizing the snapshotting of network state but continuing to rely on distributed BGP for computation of the stable forwarding tables; the snapshots simply slow down FIB changes making routers jump from one set of consistent FIBs to another.

The above body of prior work informs the design of CIRCA, a logically centralized, scalable, and fault-tolerant architecture to offload interdomain route computation as-is to the cloud without requiring ASes to reveal any additional proprietary policy information while ensuring predictably quick convergence delays, a combination of goals that to our knowledge has not been achieved before.

3 CIRCA design and implementation

CIRCA is driven by the following key design goals.

1. *Limited disclosure*: CIRCA should not require ASes to disclose proprietary policies or topology to any entity.
2. *Quick convergence*: CIRCA should ensure quick convergence unlike BGP's high tail latencies (§3.2).
3. *High availability*: CIRCA should ensure high availability despite component or network failures (§3.4.2).
4. *Route computation equivalence*: CIRCA's computed routes should match those computed by BGP or any desired decentralized and safe routing protocol (§3.4.1).
5. *BGP interoperability*: CIRCA should gracefully coexist with BGP and be incrementally deployable (§3.5).

3.1 Design overview

BGP route computation, even given centralized global topology and policy information, is a computationally hard problem [13] unless restrictive assumptions are made, e.g., under Gao-Rexford (GR) conditions, the complexity is linear in the size of the network. To our knowledge, in the absence of more restrictive assumptions than safety of routing policies, previously proposed approaches for BGP route computation are not qualitatively more efficient than simulating BGP's path exploration process, i.e., some sequence of receive/import/adopt/export activations until convergence. Even logically centralized approaches based on SMPC [17], that in addition to GR constraints restrict policies to be next-hop-based and routes to be uniquely determined by the source-destination pair, compute BGP routes by simulating BGP's path exploration. Algebraic formulations of the problem can potentially compute BGP routing outcomes more efficiently than asynchronously simulating path exploration, e.g., via a generalized Dijkstra-like algorithm [46] or iterative matrix methods that synchronously simulate generalized Bellman-Ford's [45, 46] path exploration, but only under restrictive assumptions that BGP in practice is not known to satisfy, e.g., non-neighbor-based policies like "prefer customer but disprefer routes containing AS X" are neither *left-distributive* nor *monotonic* (also known as *strictly increasing* [8]), each of which is known to be a sufficient condition for computing destination-based forwarding routes efficiently.

CIRCA's high-level design based on virtual routers replaying BGP in the cloud is naturally dictated by two premises: (1) we need to limit disclosure of proprietary policy information to no more than what is shared between ASes in today's BGP (also referred to as *ground-BGP*); (2) even with centrally disclosed policy information, without more restrictive assumptions than just safety, we don't know of a qualitatively more efficient way to compute BGP's routing outcomes other than to simulate its path exploration. Accordingly, CIRCA maps virtual router incarnates (or *avatars*) in the cloud to each *ground router*. In what follows, we first describe unreplicated CIRCA (deferring CIRCA's replication mechanisms for ensuring high availability amidst failures to §3.4.2) wherein each ground router is one-one mapped to a cloud avatar and the cloud avatars execute a protocol derived from the underlying distributed ground routing protocol.

CIRCA operates incrementally in response to *root cause events*, i.e., external events such as a node or link up/down event, link cost change, or an operator-induced policy or configuration change. CIRCA operates in three phases— (1) detection and cloudward dispatch of a root event by a ground router; (2) route computation by cloud routers; (3) adoption of forwarding outcomes dispatched by cloud routers groundward—a high-level design also shared by several prior works [2, 4, 17, 26, 41].

3.1.1 CIRCA north-south protocol

The CIRCA north-south protocol consists of three phases.

Phase I: Upon detecting a root event, a ground router R:

1. Assigns a unique label $E = [seqn, R]$ to the event, where *seqn* is a sequence number incremented by exactly one for each locally detected root cause event;
2. Appends $\langle E, iface, etype, eval \rangle$ to a local persistent log, where *iface* identifies the interface affected by the event and *etype* and *eval* are respectively the event type and value, e.g., *etype* may denote a link cost change and *eval* the value of the new link cost;
3. Sends the ground root cause (GRC) message $\langle \text{GRC}, E, iface, etype, eval \rangle$ to its cloud avatar(s) $\mathbf{v}(R)$.

Phase II: CIRCA cloud routers then take these steps:

1. Upon receiving $\langle \text{GRC}, E = [seqn, R], iface, etype, eval \rangle$, the cloud avatar $v(R)$ initiates a distributed route computation algorithm (in §3.3) for that event after it has sequentially completed the execution for all root events $\langle [k, R] \rangle$ for $k < seqn$;
2. When a cloud router $v(Q)$ impacted by E eventually receives the termination notification (as guaranteed by the liveness property in §3.3.2) for E, it
   (a) appends the FIB entries updated as a result of E to a local persistent log;
   (b) dispatches the FIB entries to ground incarnate Q;

Phase III: A ground router Q receiving FIB entries for E:

1. Appends the FIB entries to a local persistent log;
2. Applies the FIB entries for $E = [seqn, R]$ iff it has applied all FIB entries for root events $[k, R]$ for $k < seqn$;

The high-level protocol above can be optimized as follows.

Garbage collection: A ground router Q informs its cloud avatar $v(Q)$ to garbage collect its log entries for any root cause event E for which Q has already applied (in step III.2 above) the updated FIB entries. The ground router only persists $O(1)$

state per prefix. For each prefix p, it persistently maintains only the FIB entries corresponding to the root event E that it most recently applied in step III.2. §3.4 explains why this preserves Route Computation Equivalence even amidst faults.

Concurrent execution: Step II.1 can be executed in parallel for two or more root events while preserving key safety and liveness properties (as explained in §3.3.3).

Virtual router consolidation: The one-one mapping of ground to virtual routers in CIRCA can be optimized by using a single, more complex route computation server per AS that emulates the receipt/sending of eBGP updates from/to adjacent ASes. Our focus on one-one router mapping in this paper is driven by our goal to show a proof-of-concept design and implementation that works *at all* and at scale, so we defer approaches to optimize cloud resources to future work.

The rest of this section §3 describes how CIRCA achieves its design goals starting with the first limited disclosure goal.

Limited disclosure: CIRCA's threat model assumes that ASes trust the cloud hosting provider to provide the physical infrastructure in a non-curious manner, an assumption consistent with standard IaaS cloud computing industry practice. From a practical standpoint, there is a significant difference between an AS trusting a cloud provider to not peek into its virtual machines versus that AS explicitly submitting all of its proprietary policy information to the cloud provider. Limiting overt leakage of proprietary information further requires that virtual routers belonging to a single AS be hosted within a virtual LAN (VLAN) within the CIRCA cloud so that IGP, iBGP, or other intradomain messages are physically confined within the VLAN. §5 discusses state-of-the-art secure computing techniques to extend CIRCA to work even with a *honest-but-curious* cloud provider in the future.

3.2 MRAI: Unnecessary evil in the cloud

An important reason interdomain routing suffers from long convergence delays is the presence of MRAI (min route advertisement interval) timers [3, 28, 31, 36, 43, 49]. A nontrivial body of prior work suggests that MRAI timers are a necessary evil to tame BGP's message complexity. Early work by Labovitz et al. [28] suggests that without such timers, BGP's message complexity may be superexponential (or $O(n!)$). Although the gadgets exemplifying the superexponential message complexity in that work are somewhat contrived in that their policy configuration does not satisfy Gao-Rexford (GR) conditions that are believed to hold commonly in practice, subsequent work [9] has shown that BGP's message worst-case complexity can be exponential even in GR topologies.

Our position is that MRAI timers even in ground-BGP today are overly conservative and are set to high values (e.g., 30s is a commonly recommended default value [31,50,51]) in part because the relationship between MRAI timers, message complexity, and overall convergence delay is poorly understood. Classical work on this topic [12] suggests that convergence delay exhibits a nonmonotonic relationship to MRAI timers, i.e., there is a minima or a sweet spot setting for the timer below which operators risk worsening convergence delay because of prohibitive message complexity and above which the timers themselves are too conservative exacerbating delay. There isn't universal agreement on what value of the timer is optimal; some have suggested that the common default of 30s is too high [12, 19, 31] while others have noted the risks of heterogeneous timer values further exacerbating message complexity [9], so conventional operational wisdom has been to err on the conservative side.

3.2.1 Why the cloud is different

There are three critical differences between the cloud and ground-BGP with implications for MRAI and other timers.

1. *Control bandwidth*: The CIRCA cloud can be easily provisioned with 2-3 orders of magnitude more control bandwidth (e.g., tens of Gbps) compared to typical control traffic provisioning in ground-BGP today.
2. *Propagation delay*: The propagation delay diameter of the CIRCA cloud is 2-3 orders of magnitude less than the hundreds of milliseconds in ground-BGP today.
3. *Control-data isolation*: Interference between control and data is a non-concern in the CIRCA cloud, so it can afford to be much more wasteful than ground-BGP.

Accordingly, we pick an aggressive point in the design space for CIRCA, namely, no MRAI timers at all! The justification for this design choice is as follows. First, MRAI timers, when they help, intuitively work by limiting spurious path exploration by making each node wait long enough to hear from its neighbors about their reaction to earlier routing messages. However, these very timers can make nodes unnecessarily wait for several MRAI rounds even when there is no possibility of excessive spurious path exploration. Worse, some (admittedly contrived) choices of heterogeneous timer values even in GR settings can actually *cause* exponential message complexity [9] even though overall convergence delay in GR settings is determined by the speed of information propagation along the customer-provider chain and back. Eliminating MRAI timers altogether propagates information as quickly as possible while naturally slowing down node reaction times because of message queuing delays in the rare cases that message processing is indeed a bottleneck.

Second, message processing delays in commodity routers as well as processing delays in well provisioned cloud environments are an order of magnitude lower than the numbers used in prior simulation-based work [12, 43]. Third, we hypothesize that reasonable values of MRAI timer values when useful are positively correlated with propagation delays, i.e., the higher the propagation delays, the higher the timer values needed, all else being equal. This hypothesis is consistent with (but not strictly implied by) the simple model in [37].

Our prototype-driven experiments in §4.1 validate the hypotheses above showing both that MRAI timers are unnecessary in reasonably provisioned cloud settings and that our results do not qualitatively contradict the nonmonotonic behavior reported in previous works when compute provisioning is artificially limited to unrealistically low levels.

3.3 Distributed convergence detection

The description thus far has side-stepped a critical question: *How do* CIRCA *cloud routers replaying a distributed protocol like BGP know when the processing of a root cause event has converged?* A convergence detector is necessary for a cloud router to initiate transmission of the corresponding updated FIB entries to its ground incarnate. The distributed BGP protocol as executed by Internet routers today does not have any such indicator of convergence (other than, trivially, the absence of any routing updates for a sufficiently long time).

To appreciate why detecting convergence in CIRCA is not easy, consider the strawman strategy of each virtual router periodically, say once every τ seconds, reporting the most recent time when it processed a message for a root cause event e to a centralized monitor that declares convergence when no router has processed any message for e in its most recent reporting interval. For $\tau = 5$s and a total of say 10^6 routers, this reporting traffic alone for each root event is 200K pkts/s. A hierarchical convergence detector that employs a per-AS detector with 50K ASes will still incur a reporting traffic of 10K pkts/s per root event, which is prohibitively expensive especially given that many root events may have only a localized impact affecting the forwarding behavior of a small number of ASes if at all. More importantly, a centralized monitor will incur a convergence delay of at least τ seconds per root event, not to mention poor fault tolerance. Finally, a centralized monitor by design observes more information than can be observed or easily inferred by any AS in BGP today, thwarting CIRCA's limited disclosure goal (§3.0).

We present a completely distributed convergence detector that is provably safe, i.e., no false positives, and live, i.e., it will detect convergence provably within at most 3× the actual convergence delay—and in practice within a much smaller factor (§4.2)—and the number of BGP messages as observed by a (theoretical) global monitor. The distributed convergence detection algorithm works in three phases as explained below. For ease of exposition, we first assume failure-free execution (deferring fault tolerance to §3.4). The algorithm is bootstrapped with a set of path updates generated by simulating the root event at the "root" cloud router receiving the corresponding ground root cause message.

Figure 2 illustrates the three phases of the convergence detection algorithm. In the first exploration phase, for each root event e processed by a virtual router, the router maintains state about the event by adding it to a set of unconverged events. When a virtual router R processes a BGP message m related to e and it induces no change to R's FIB (and consequently no new announcements to its neighbors), we say that message m "fizzled" at R. For each fizzled message, R back-propagates

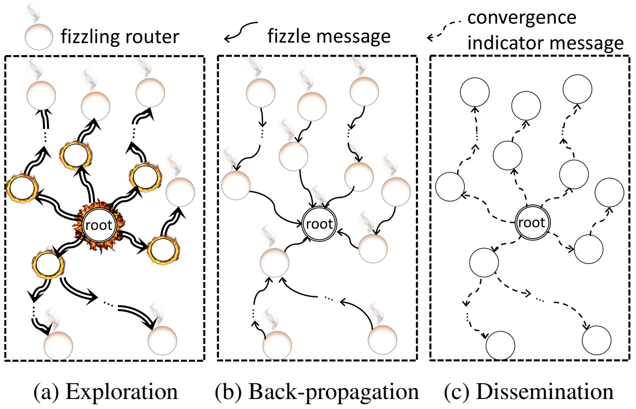


Figure 2: 3-phase distributed convergence detection.

a fizzle indicator to its peer that previously sent it m thereby initiating the second back-propagation phase. Each router locally keeps track of each message pertaining to e that it sends to each neighbor and waits for the corresponding fizzle acknowledgment to be back-propagated. When the router that initiated the root cause event has received fizzle acknowledgements for all messages it originated, it determines e to have converged, which initiates the third dissemination phase wherein, starting with the root router, each router forwards the convergence indicator along each link previously traversed in the exploration phase, at which point it drops all state related to e for convergence detection.

3.3.1 Formal event-action protocol

In order to formally reason about the safety and liveness properties of the distributed convergence detector, we codify it in event-action format in Algorithm 1. The key notation is as shown in Table 1. Where there is little room for confusion, we drop the argument R implicitly fixing it to *self* at a router.

Algorithm 1 shows how a router R handles three distinct events: (1) receipt of a root cause message $\langle \text{GRC}, E \ldots \rangle$ from its ground incarnate $v(R)$ that initiates path exploration; (2) receipt of a CBGP message from a peer cloud router; and (3) receipt of a FIZZLE message from a peer cloud router.

The first event handler processes the received root cause message by "simulating" the corresponding ground event, which produces a set of resulting update messages announcing or withdrawing a set of paths to affected prefixes, a set denoted as *paths(E)*. If the root event E does not change R's FIB, then routing has trivially converged (line 7). Else, for each changed entry in R's FIB causing a new update subject to its export policy, it creates a unique timestamp as a two-tuple consisting of a strictly increasing logical clock returned by *now()* and the router's identity. (This logical clock does not need to reflect the happens-before relationship between events like Lamport clocks for reasons that should be clear from the formal proofs in §A.) The router stores each resulting update to its peers in a map $sent[E]$ and remembers the cause of each sent update

Algorithm 1 Distributed convergence detection (DCD) and route computation at cloud router $v(R)$

```
1: event RECV(⟨GRC, E, iface, etype, eval⟩, R):   ▷ upon receipt of root cause message from ground router R
2:     $paths(E) \leftarrow sim(\langle \text{GRC}, E, iface, etype, eval \rangle)$
3:     $FIB_0 \leftarrow FIB$
4:     for each prefix set $p$ in $paths(E)$ do
5:         $FIB \leftarrow BGPImport(FIB, \langle \text{CBGP}, E, p, paths(E)[p] \rangle)$
6:     if $FIB = FIB_0$ then
7:         $converged(E) \leftarrow true$; exit
8:     else
9:         $ts \leftarrow [now(), R]$
10:        for each $r$: $BGPExport(FIB_0, FIB)$ do
11:            $ts_1 \leftarrow [now(), R]$
12:            $sent[E][ts] \cup= [r.peer, r.prefixes, ts_1]$
13:            $cause[ts_1] \leftarrow \langle \text{GRC}, E, ts, v(R) \rangle$
14:            $send(\langle \text{CBGP}, E, r.prefixes, r.asPath, ts_1 \rangle, r.peer)$
15: ________________________________________
16: event RECV(⟨CBGP, E, p, asPath, ts⟩, N)   ▷ upon receipt of CBGP message for prefixes p from cloud peer N
17:     $FIB_0 \leftarrow FIB$
18:     $FIB \leftarrow BGPImport(FIB, [\text{CBGP}, E, p, asPath])$
19:     if $FIB = FIB_0$ then
20:         $send(\langle \text{FIZZLE}, E, ts \rangle, N)$
21:     else
22:         for each $r$: $BGPExport(FIB_0, FIB)$ do
23:             $ts_1 \leftarrow [now(), R]$
24:             $sent[E][ts] \cup= [r.peer, r.prefixes, ts_1]$
25:             $cause[ts_1] \leftarrow \langle \text{CBGP}, E, ts, N \rangle$
26:             $send(\langle \text{CBGP}, E, r.prefixes, r.asPath, ts_1 \rangle, r.peer)$
27: ________________________________________
28: event RECV(⟨FIZZLE, E, ts⟩, N)   ▷ upon receipt of a fizzle message from cloud peer N
29:     $ts_0 = cause^{-1}(ts).ts$
30:     $fizzled[E][ts_0] \cup= sent[E][ts_0][ts]$   ▷ for dissemination phase
31:     $sent[E][ts_0] -= sent[E][ts_0][ts]$
32:     if $sent[E] = \{\} \wedge cause^{-1}(ts) = \langle \text{GRC}, E, \ldots \rangle$ then
33:         $converged(E) \leftarrow true$; exit   ▷ begin dissemination phase
34:     else if $sent[E][ts_0] = \{\}$ then
35:         $send(\langle \text{FIZZLE}, E, ts_0 \rangle, cause^{-1}(ts).peer]$
```

CBGP	Message type of BGP messages exchanged by cloud routers
GRC	Message type of root cause messages sent by a ground router
v(R)	Cloud incarnate of ground router R
paths(E)	Paths affected by E, a unique root cause label, to one or more prefixes
FIB(R)	Forwarding table of router R
send(m, N)/*recv*(m,N))	send/receive message m to/from peer N
BGPImport(F,m,R)	New FIB resulting from processing message m at router R with FIB F
BGPExport(F_1,F_2,R)	Announce/withdraw messages by R, filtered by its export policy, upon a FIB change from F_1 to F_2

Table 1: Notation used by Algorithm 1.

as the original GRC message in a *cause* map indexed by the timestamp *ts* of the sent update. By definition of *now()*, each sent update has a unique timestamp.

The second event handler, the common-case action invoked at a cloud router beyond the root router, is similar to the first but with two important differences. First, if a received update $\langle \text{CBGP}, E, p, asPath, ts \rangle$ from a peer does not change its FIB, it responds with the corresponding $\langle \text{FIZZLE}, E, ts \rangle$. Second, if it does change its FIB, it remembers the cause of each resulting export-policy-filtered update as the incoming update.

The third event handler purges entries from the *sent*[E] map upon receipt of the corresponding fizzle messages (removing the message with timestamp *ts* from the set $sent[E][ts_0]$ in line 31). If the *sent* set caused by an update is emptied at a non-root router, it back-propagates a fizzle to the peer that sent the incoming causal update (line 35). When an incoming fizzle message empties the *sent*[E] map at the root router (line 33), it declares the distributed processing of event E as converged, and initiates the third dissemination phase (deferred to §A).

We need to formally prove the correctness property that when a root router thinks routing has converged, that is indeed the case; and the liveness property that the above protocol will eventually terminate under realistic conditions.

3.3.2 Safety, liveness, and efficiency

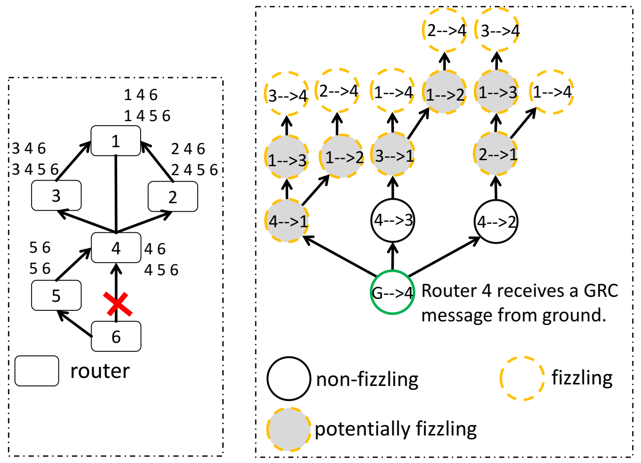
The formal proofs of all claims herein are deferred to §A.

Theorem 3.1. SAFETY*: If converged(E) is true at the root cloud router that received the* GRC*, then no further* CBGP *message for E will be received by any cloud router.*

Theorem 3.2. LIVENESS*: Algorithm 1 eventually terminates, i.e., converged(E) is set at the root router, if BGP is safe*[1] *and all cloud routers are available for sufficiently long.*

The safety and liveness proofs rely on a construction called the *directed message graph* produced by any execution instance of Algorithm 1, as illustrated in Figure 3, wherein each vertex corresponds to a message—either the original GRC or a CBGP message—and there is a directed edge from a message m_1 to another message m_2 if m_1 *caused* m_2, i.e., m_2 was sent in lines 14 or 26 in response to the receipt of m_1 in lines 1 or 16 respectively. We say that $m_1 \rightarrow m_2$ (read as m_1 *happened before* m_2) if there exists a directed path from m_1 to m_2 in the graph. It is straightforward to show (Lemma A.1) that the directed message graph is a directed tree, i.e., it is acyclic and each vertex has exactly one incoming edge. The proof of safety relies on showing that if $m_1 \rightarrow m_2$, then m_1 fizzles only after m_2 has previously fizzled (Lemma A.3) and when *converged*(E) is true at the root router, no CBGP messages for E are under propagation anywhere (Lemma A.4). The proof of liveness relies on showing that, with safe BGP policies, every CBGP message eventually receives a matching FIZZLE (Lemma A.5) as well as on Lemma A.3 and on the assumption

[1]"BGP is safe" means that it is guaranteed to converge to a stable route configuration [13] and is unrelated to *safety* in the theorem just above.



(a) Routing topology (b) Message DAG

Figure 3: (a) Routing topology: root cause is the failure of link 4-6, arrows are from customer to provider, and no arrows means a peer relationship. (b) Potential message DAG evolution: an execution instance of Algorithm 1 produces a prefix of the shown directed tree where dashed-greyed (yellow) circles may or may not be a fizzling message, dashed (yellow) ones are fizzling, and solid ones are non-fizzling messages.

that a router records all state changes in a local persistent log before sending messages based on the changes.

For concision, we defer a codification of Algorithm 1's *dissemination* phase to §A.3. This phase does not impact safety or liveness as defined above but is needed to show the stronger liveness property that the algorithm terminates for all impacted cloud routers (not just the root).

Theorem 3.3. EFFICIENCY*: The root router (Any router) in Algorithm 1 detects convergence within at most* 2Δ (3Δ) *time and* $2M$ $(3M)$ *messages where* Δ *and* M *are respectively the actual convergence delay and number of messages incurred by the distributed route computation in the cloud.*

Although a $3\times$ delay overhead may seem high and it may cursorily appear possible to reduce that overhead with simple optimizations, our attempts at doing so while preserving provable safety and liveness have been elusive. Fortunately, our experiments (§4.1) show that (1) convergence delays in the cloud are significantly smaller than those in the ground, so a $3\times$ overhead in the cloud is still a big net win; and (2) the overhead is much smaller than $3\times$ because messages in the first exploration phase can contain a large number (even thousands) of prefixes, but messages in the other two phases are like small acknowledgments.

3.3.3 Concurrent event processing

The discussion above implicitly focused on sequentially processing one root event at a time. However, we can not afford the luxury of a centralized monitor or other mechanisms to ensure sequential processing of root events for the same reason that convergence detection had to be distributed in the first place. With multiple concurrent root events being processed by different cloud routers, Algorithm 1 in conjunction with the high-level end-to-end protocol in §3.1.1 has a problem: the FIBs dispatched by a cloud router in step II.2.b may be inconsistent as they may reflect the incomplete processing of one or more root events being concurrently processed in the system, which in turn could result in transient loops or blackholes in the data plane at ground routers (as can happen in ground-BGP today even with just a single root event). However, the safety and liveness properties above as well as Route Computation Equivalence as formalized in the next subsection still hold, so our CIRCA implementation simply processes concurrent root cause events in parallel. A more detailed discussion of the pros and cons of concurrent event processing while preserving route consistency in the ground data plane is deferred to a technical report [38].

3.4 Route Computation Equivalence despite link or router failures

Informally, this section shows that any decentralized policy routing (or "ground") protocol, including but not necessarily limited to BGP, satisfying a naturally desirable consistency property (ECC, as formalized below) can offload its route computation to CIRCA with the guarantee that the CIRCA-equipped system will eventually achieve the same routing outcomes as the unmodified ground protocol despite failures, a property referred to as Route Computation Equivalence. Unlike the safety and liveness properties shown for CIRCA's cloud *control plane*, the results in this section subsume the *data plane* or end-to-end forwarding behavior on the ground.

Network state model. We model the ground network as a state machine with root cause events effecting state transitions. The *state* of the network encompasses the (up/down) state of all links and routers as well as any configuration information (link costs, operator-specified policies at a router, etc.) that potentially impacts routing outcomes. Let S_0 denote the initial state of a network and $[e_1,\ldots,e_k]$ denote a sequence of root cause events such that each event e_i transitions the network from state S_{i-1} to state S_i. The state of the network after an event or a sequence of events is denoted using the operator '$|$' as in $S_1 = S_0|e_1$ or $S_k = S_0|[e_1,\ldots,e_k]$.

Definition 1. EVENTUALLY CONSISTENT CONVERGENCE (ECC)*: If no root cause events occur for a sufficiently long period, forwarding behavior should converge to reflect the state of the network just after the most recent root cause event.*

We posit *eventually consistent convergence* as defined above as an intrinsically desirable property for any practical routing protocol. ECC as defined is rather weak because "eventual" allows routes to be inconsistent with global network state for arbitrarily long, but BGP today does satisfy this property provided routing policies are safe (or convergent) and every event is eventually detected and acted upon by incident routers immediately impacted by it.

3.4.1 Unreplicated CIRCA ensures RCE

We next formally define Route Computation Equivalence.

Let $\mathbf{D}(S)$ represent any distributed route computation function that, given network state S, returns a set of possible *routing outcomes*, i.e., a set $\{\mathbf{GFIB}_1, \mathbf{GFIB}_2, \ldots\}$ wherein each element represents global forwarding behavior as captured by the union of the FIBs of all routers in the network. The reason $\mathbf{D}(S)$ is a set of size greater than one is to incorporate non-determinism as BGP even with safe policies in general can have multiple stable routing configurations, e.g., the DISAGREE gadget [13] converges to one of two possible stable routing outcomes depending on node reaction times. Let $\mathbf{GFIB}(t)$ denote the forwarding routes adopted by ground routers at time t. We would like to show that if a distributed route computation process satisfies ECC, CIRCA preserves those ECC routing outcomes. Formally,

Definition 2. ROUTE COMPUTATION EQUIVALENCE (RCE)*:* Given an initial network state S_0 and a finite sequence of root case events $[e_1, \ldots, e_n]$, a cloud-assisted routing protocol is said to preserve equivalence with respect to a distributed route computation function $\mathbf{D}(S)$ if it ensures that $\lim_{t\to\infty} \mathbf{GFIB}(t) \in \mathbf{D}(S_n)$ where $S_n = S_0|[e_1, \ldots, e_n]$.

Next, we show that a single-datacenter (or unreplicated) pure CIRCA deployment ensures RCE despite intermittent failures (proof deferred to Appendix B), where *pure* means all ground routers have been upgraded to rely only on CIRCA.

Theorem 3.4. *If for a sufficiently long period—(i) all ground routers can reach a* CIRCA *cloud replica and vice versa; and (ii) all cloud routers are available and can communicate in a timely manner—a pure* CIRCA *system ensures Route Computation Equivalence with any distributed route computation function that satisfies Eventually Consistent Convergence.*

3.4.2 Replicated CIRCA ensures RCE

To see why Theorem 3.4 holds even in a replicated CIRCA deployment without any replica coordination protocol, we simply observe that each CIRCA replica independently ensures RCE, i.e., the FIBs it computes reflect the most recent state of the network provided it eventually receives all root cause event reports (in any order). If each ground router is responsible for relaying each root event to all of its cloud avatars, no replica coordination protocol between CIRCA replica sites is necessary. CIRCA cloud routers may optionally relay root events to its siblings on other replica sites as an optimization, but this is not necessary for ensuring Route Computation Equivalence. An analogous observation, namely that no sophisticated replica coordination protocol is needed for safety, has been long known for a single-domain route computation service (e.g., RCP [4]), but not for interdomain routing. Furthermore, the technical reasons why they hold in the CIRCA architecture based on processing root cause events consistently are very different from RCP that relied on an assumption of consistent views of the ground network despite partitions across replicated route servers.

Overhead of faults. CIRCA's simple design maintains RCE despite arbitrary failure patterns but failures, specifically of cloud routers or link failures inducing ground-cloud unreachability, have two costs: (1) growing log of unprocessed root events at ground routers, and (2) transient forwarding loops or blackholes. As detailed in the techreport [38], the former is bounded by the size of the total configuration state at a router and the latter can be alleviated (but not eliminated) by replicating CIRCA datacenters in a *pure* CIRCA deployment or by relying on BGP co-existence mechanisms below.

Conveniently, ground router failures are a non-issue because of the fate sharing property that it is the only router stalled by its failure; neighboring ground routers will detect and process its failure as a normal root cause event.

3.5 Co-existence & incremental deployability

Co-existence refers to the ability of ground routers to leverage the CIRCA cloud while continuing to rely on ground-BGP as a fallback or an optimization under the (possibly impractical) assumption that all ground routers have been upgraded to be CIRCA-capable. *Incremental deployability* refers to the ability to gainfully deploy CIRCA despite upgrading only a subset of ground routers to be CIRCA-capable. (As defined, a deployment can not be both co-existent and incremental.)

3.5.1 Co-existence with ground-BGP

CIRCA's support for co-existence enables ground routers to get the best of both worlds, i.e., adopt forwarding outcomes from whichever plane—ground or cloud—converges earlier, thereby also implicitly relying on ground-BGP alone as a fallback when the cloud is unreachable. To this end, ground-BGP needs to be extended so as to tag each message with the corresponding root cause event label, and ground-BGP routers need to relay the tag in any incoming message by inserting it into any outgoing messages caused by it.

A ground router R uses the root event label $E = [seqn, R]$ in a ground-BGP message as follows. If R has already received $\Delta FIBEntries(E)$ for E from the cloud, it rejects any further ground-BGP messages tagged with E from impacting its FIB, otherwise it continues processing ground-BGP messages as usual through its decision engine with one difference: it marks any changes to its FIB as a result of E as such and keeps a copy of the original entry for the corresponding prefix, which essentially allows R to undo the effect of any ground-BGP messages tagged with E. When R eventually receives $\Delta FIBEntries(E)$ from the cloud, irrespective of whether or not ground-BGP has already converged at R (necessarily unbeknownst to it), it installs $\Delta FIBEntries(E)$ anyway undoing the effect of any ground-BGP messages pertaining to E.

Co-existence further requires ground routers to maintain a route information base (RIB), or the set of available routes, unlike universal CIRCA (wherein every router is CIRCA-capable) that only required ground routers to maintain a FIB (or the set of *adopted* routes). Maintaining a RIB plus a FIB is no different from ground-BGP today, and requires ground routers to continue processing messages tagged with E so as to update its RIB (but not its FIB) even after it has received the corresponding $\Delta FIBEntries(E)$ from its cloud incarnate. Maintaining a RIB in co-existent CIRCA also enables ground routers to employ fast reroute or backup routing options during the ground or cloud convergence period.

3.5.2 Incremental deployability

Incremental deployment means that some ground routers may be completely CIRCA-unaware, so such legacy routers can not even parse BGP messages with root cause labels. In such scenarios, the benefits of CIRCA would be limited to root cause events that originate within and whose exploration fizzles entirely within a contiguous island of routers all of which have been upgraded to be CIRCA-capable. For events that entirely fizzle within the contiguous upgraded island, the protocol is identical to the co-existent scenario (§3.5.1). If an event spills out of this island, it is recognized as such by any CIRCA cloud router at the island's boundary. When a cloud router detects a spill-over, it immediately aborts the exploration by back-propagating an ABORT message (instead of a FIZZLE), enabling the root router to conclude that the CIRCA cloud can not compute the outcome of that event after all.

CIRCA needs an additional mechanism in order to preserve RCE in incremental deployment scenarios, in particular, to ensure that cloud routers do not diverge from their ground counterparts. To this end, each *boundary router* in the CIRCA cloud, i.e., any router whose ground incarnate has at least one neighbor that has not been upgraded to be CIRCA-capable, establishes receive-only ground-BGP sessions with those (ground) routers, wherein a *receive-only* session is one that only receives, but never sends, any announcements. Note that, notwithstanding this additional peering requirement, because of the non-deterministic nature of ground-BGP, it is possible for some events to (not) fizzle within the upgraded island in the cloud even though they might have not fizzled (fizzled) within the corresponding island on the ground. §B formally shows why RCE is nevertheless preserved under incremental CIRCA deployment scenarios.

3.6 CIRCA implementation

We implemented a Quagga-based prototype of CIRCA including the complete north-south protocol and the distributed convergence detection algorithm as described in roughly 2,430 lines of code. We have not implemented co-existence or incremental deployability mechanisms (§3.5) but evaluate their benefits via simulations based on real AS topologies.

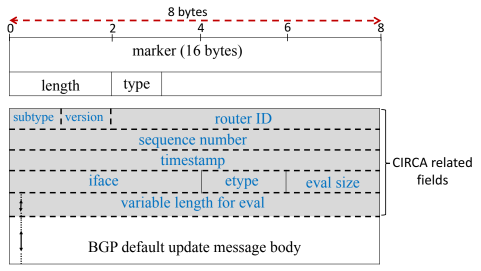


Figure 4: Header and body of a CIRCA BGP packet

Figure 4 shows the format of header and body of a CIRCA packet. The header is just the BGP header as also implemented in Quagga. The new fields are shown in gray background and are as follows. The subtype identifies the internal type of a CIRCA packet and include message types such as GRC, FIZZLE, and CONVERGED used in the convergence detection protocol. The `router ID` and `sequence number` two-tuple uniquely identify a root cause event E in Alg.1; `timestamp` is a unique ID for each message in Alg.1; and `iface`, `etype`, and `eval` are as described in §3.1.1. The rest of the packet is just the body of a BGP update packet with path attributes and network layer reachability information.

4 Evaluation

In this section, we conduct a prototype- and measurement-driven evaluation of these questions: (1) Does CIRCA help significantly drive down convergence delays by being "care-free" about message complexity? (2) Does CIRCA help improve end-to-end convergence delays including (a) the overhead of distributed convergence detection and (b) north-south communication delays? (3) Does CIRCA yield incremental benefit in incremental deployment scenarios?

All prototype-driven experiments are conducted on an Emulab LAN testbed of 150 physical machines each with 8 cores. All interfaces on used physical machines are connected to gigabit ports on HP ProCurve switches 5412zl series. We use three experimental-net switches (procurve3-procurve5), each with approximately 240 ports. A fourth switch (procurve1) acts as the center of a "hub and spoke" topology for the experimental network. The link speed between each pair of machines is 1Gbps and that of each virtual interface is 10Mbps.

4.1 Impact of MRAI timers

4.1.1 Single-router AS setup

In order to evaluate the convergence delay vs. message complexity tradeoff in the CIRCA cloud, we conduct a prototype-driven evaluation on the Emulab testbed using different realistic Internet topologies extracted from 2018 CAIDA AS-level topology gathered from RouteViews BGP tables [42] and varying MRAI timer values. Our topology includes 60,006 ASes and 261,340 links with inferred relationships between linked ASes [11]. Because of our limited physical resources

Number of routers in topology	20	60	180	540	1200
Average degree of routers	4	8	13	19	29
Average number of prefixes	225	296	180	155	106
All unique prefixes	4.5k	17.8k	32k	82.7k	125.3k

Table 2: Properties of used topologies.

(150 physical machines with 1200 cores), we extract contiguous subgraphs of the AS topology of varying sizes from 20 to a maximum of 1200 ASes. For extracting a subgraph of n ASes from the CAIDA data set, we first randomly pick an AS and perform a breadth-first search until the number of selected nodes reaches n. We increase the network size by only adding new nodes to the smaller subgraphs. The average degree of nodes and the number of prefixes belonging to each AS in our extracted topologies are shown in Table 2.

Our experiments retain Quagga's so-called "burst" MRAI timer implementation where all updates to a given peer are held until the timer expires, at which time all queued updates are announced. Quagga applies the MRAI timer only on updates, not on withdrawals, a behavior we retain. The published official BGP4 protocol specification RFC 4271 [39] suggests that route withdrawal messages are also to be limited by the MRAI timer; a change from earlier version (RFC 4098) where withdrawals could be sent immediately [40].

We emulate four different root cause events in our experiment; node up, node down, link up and link down. For each network size and MRAI value, we first assign the routers in the network to physical machines randomly and set the same MRAI timer on all routers and then wait for a long enough duration until routes to all prefixes at all routers stabilize. At this point, we randomly pick a router in the topology and trigger a random root cause event, and repeat this process 30 times for each topology size and MRAI value. We log each BGP message and post-process log files from all routers at the end of each run to compute the convergence delay.

Figure 5 shows the average convergence delay of 30 simulated root cause events using different MRAI values. In computing the convergence delay of each root event, we do not count the time it takes for the root router to detect the event because this delay ($\approx$ one to few seconds) will be incurred by ground routers but not CIRCA cloud routers. We consider the time when the last immediately impacted incident router detects the event as the beginning of the convergence period.

A couple observations are noteworthy. First, the absolute values of the convergence delay with zero MRAI timers are small. Convergence delay increases as expected with the topology size and number of affected prefixes of root router and the highest (average) value observed is 2.3 seconds with 1200 ASes. We conduct a smaller-scale experiment with multiple routers per AS using realistic intradomain topologies (deferred to a techreport [38]). Unsurprisingly, the qualitative trend of increasing convergence delays with increasing iBGP

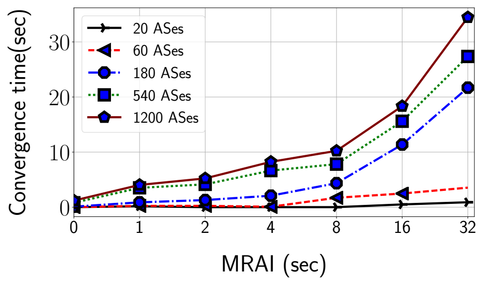


Figure 5: Average convergence delay of BGP across all root events with different MRAI timers on Emulab.

MRAI values persists. Second, convergence delay monotonically increases with the MRAI timer value and a zero timer yields the lowest convergence delay. This observation is in contrast to an established body of prior research [9, 12, 28, 43] suggesting a non-monotonic relationship between MRAI timer values and convergence delays and the existence of a minima for convergence delay at an MRAI value greater than zero, and that lower or zero MRAI values can significantly exacerbate convergence delays.

There is no contradiction however with prior findings. Previous work has been largely based on simulations and toy topologies (e.g., [12, 43]) and assumed unrealistically high values of message processing delays, e.g., random delay from zero to 1 second for each message in [12] and a uniformly distributed delay between 1 and 30 milliseconds for message processing [43]. There is no fundamental reason for message processing delays to be that high on modern processors and we find that they are at best tens of microseconds in Quagga (after we systematically disabled unnecessary timers).

4.1.2 Reconciling prior findings

As a sanity check, to verify that with artificially inflated message processing delays, we can reproduce the nonmonotonicity observed in prior work, we reduce the number of physical machines in our Emulab setup from 150 to 50 machines and rerun the experiment with the largest topology size (1200). With this setup, each 8-core machine now has almost 24 Quagga instances running on it compared to roughly eighth Quagga instances per machine (or one Quagga instance per core) in the earlier setup. Quagga is single-threaded software, so a Quagga instance can not leverage other cores on a physical machine even if they are idle.

Figure 6 shows that the nonmonotonic behavior is reproducible with more compute stress. However, it is important to clarify that this behavior is not solely because of the modest 3× decrease in resource provisioning. Because Quagga is a single-threaded software, an MRAI timer of 0 causes it to spin in a busy wait consuming 100% CPU utilization on each core even with one core per Quagga instance, a problem that exacerbates with 3-4 Quagga instances per core. Without this implementation artifact of wasteful busy waiting, we were unable to reproduce the nonmonotonic behavior observed in

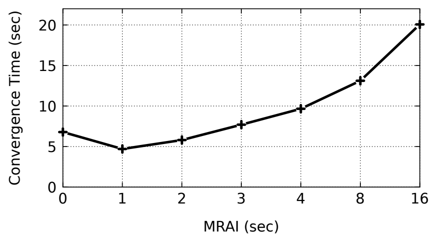


Figure 6: Non-monotonic trend of convergence delay vs. MRAI timer value with the 1200-AS topology.

prior work. Our observations therefore suggest that, with modern processing fabrics in routers, it may be worth revisiting whether conservatively set high MRAI values are appropriate even in ground-BGP today.

4.1.3 Extrapolating our findings to 60K ASes

Can even larger topologies going all the way up to 60K ASes result in prohibitively high message complexity exacerbating convergence delays? We think it unlikely for several reasons.

First, most root events are unlikely to impact a very large number of ASes unless a prefix is rendered altogether unreachable [36]. Second, even if an event impacts all ASes, with realistic Gao-Rexford policies, although pathological cases of exponential message complexity are theoretically possible, they require contrived topologies and unrealistic assumptions about relative speeds of routers to manifest in practice [22]. Even so, it may be possible to manage resource more effectively, e.g., we show in an experiment (deferred to a techreport [38]) that there is room to improve the core utilization by 30× in our testbed with multi-threading and careful mapping of virtual routers to cores. Third, even if many routers get affected by some root cause events, it is unlikely that the FIB entries (as opposed to just RIB entries) on most of the routers will change. Fourth, if message complexity does become prohibitive, with ultra-low LAN propagation delays in the CIRCA cloud, queued messages can be batch-processed (with no intervening exports) so as to provide a self-clocking mechanism to slow down announcements similar in spirit to MRAI timers but without relying on conservative, static values. This idea is similar to the proposed adaptive or dynamic MRAI values wherein the MRAI value is increased if there is more "unfinished work" in the queue and decreased if not [32, 43].

4.2 End-to-end convergence delay

In this section, we evaluate end-to-end convergence delay including the overhead of distributed convergence detection and north-south communication delays for a root cause event.

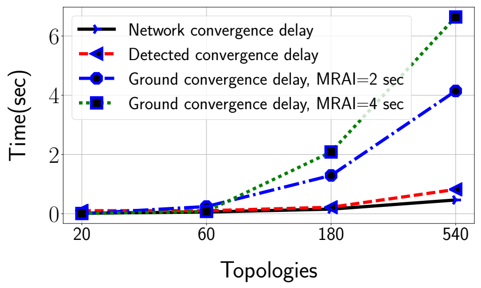


Figure 7: Difference between actual convergence delay in cloud, convergence detection delay using our algorithm, and ground convergence delay.

4.2.1 Convergence detection overhead

First, we evaluate the overhead introduced by the distributed convergence detection algorithm (Algorithm 1) to detect convergence after the protocol actually converges in cloud. As before, we estimate the ground-truth convergence delay by processing router logs and compare it to the time reported by our algorithm and repeat each root event injection 20 times.

We conduct this experiment with 20, 60, 180, and 540 ASes respectively in the ground and cloud setups with 20 randomly simulated root cause events in the ground setup in isolation. On ground, we set MRAI 2 and 4 seconds. Each ground router has a connection with its avatar in the cloud and both advertise the same list of prefixes in the network. We consider three different root cause event types in the ground network; node down, link up and link down. We have a single router AS setup for this experiment. The average degree of nodes in our topology and the number of prefixes belonging to each AS are shown in Table 2.

Figure 7 shows the average of the detected convergence time and the ground-truth convergence delay in cloud and also on ground for all root cause events for different topology sizes. Per design, we might expect the root router to take roughly 2× the time to detect and for the last router to take up to 3× time, but the observed overhead is much smaller. The reason for this fortuitous discrepancy is that messages in the first exploration phase usually contain a large number (even thousands) of prefixes, which increases processing and transmission times in that phase, but not in the other two phases.

4.2.2 End-to-end delay in getting new FIB entries

In this experiment, we evaluate the end-to-end delay for obtaining new FIB entries from the cloud.

We assume the ground routers in an island with different sizes have been upgraded to CIRCA. We run the ground instance of the routers and their avatars in a LAN (our Emulab CIRCA testbed). While we can find the convergence detection time of each router using our implemented CIRCA system, for measuring ground and cloud communication delay, we can not use the delay between ground routers and their avatars in the LAN as real Internet delay. We can estimate the

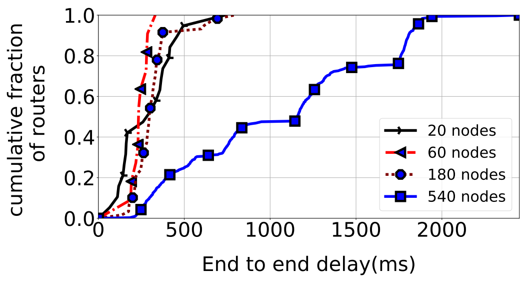


Figure 8: CDF on the end-to-end convergence delay for routers in different topology sizes

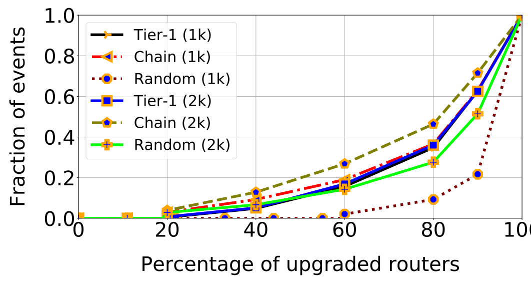


Figure 9: CDF on fraction of root cause events fizzled inside the island of upgraded routers using different approach in an island of 1,000 (1k) and 2,000 (2k) ASes

delay from Internet routers to their avatars by pinging routers in target ASes at the ground from their avatars at the cloud. The prefixes of each AS on the Internet are identifiable from the Routeviews data set [42]. For five original prefixes of our target ASes, we generate an IP address for each prefix by adding 100 to the end of the prefix, e.g., 10.10.8.100 for the prefix 10.10.8.0/24. We ping target IP addresses three times and get the average ping delay from that AS to our data center. For the roughly 20% of IP addresses for which we do not get any result, we assume 100 ms as the one-way delay.

We conduct this experiment for varying topology sizes with 20 randomly simulated root cause events on the ground in isolation. We consider 20, 60,180, and 540 ASes on the ground and in the cloud.

Figure 8 shows the end-to-end delay of getting the first entry in the FIB table to ground routers in an island with different sizes after simulating the root cause event on the ground. The end-to-end delay is the sum of the delay in sending event data to the cloud (approximated crudely as the ping delay), detecting the stable state of the network by each router, and receiving the new FIB entries from the cloud (ping delay) across all root cause events and routers affected by our root cause events. In small topologies, most of the routers (80%) get the FIB entry across all root cause events in less than 400 ms. However, for our biggest topology size, 540 nodes, we have 1.5 seconds for around 80% of routers. As explained in section 4.1.3, most of the delay is because of the delay in the first phase of our convergence detection algorithm and could be further optimized.

4.3 Incremental deployment of CIRCA

We evaluate the fraction of root cause events that fizzle entirely within an island of upgraded single-router ASes chosen as a subset of the Internet's AS topology. We sequentially simulate 20 randomly chosen root cause events originating within each upgraded island. We consider different approaches for selecting the upgraded routers: (1) *tier-1* that first "upgrades" the contiguous ASes that do not have any provider and then their customers, and so on; (2) *random* that picks ASes in the network randomly (possibly in a non-contiguous manner); and (3) *chain* that picks a random contiguous chain of customer-provider ASes. We do not consider stub ASes in our experiment as candidate ASes for upgrading.

Figure 9 shows the fraction of root cause events that fizzle entirely within upgraded islands of different sizes with different approaches. For example, for covering 40% of all root cause events, we need to upgrade roughly 80% of routers with the *chain* or *tier-1* approaches.

5 Discussion: Security, privacy, open issues

Our first stab at a practical interdomain-routing-as-a-service system leaves open questions that merit further discussion.

Any new system inevitably introduces new vulnerabilities. Our high-level goal in this work was to limit misbehavior to what is already possible in BGP's control plane, and to limit information disclosure explicitly by design, however CIRCA itself introduces new security vulnerabilities as well as side-channel information leakage as discussed below.

Side channel leakage. First, CIRCA allows a rogue AS to exploit rapid convergence as a probing mechanism to infer policies of other ASes by willfully tampering its announcements and observing how others react, an attack also possible in "slow motion" in ground-BGP, and mechanisms similar to route flap damping may be necessary in the CIRCA cloud if such "fast-forwarded" probing attacks were to become a credible threat. FIZZLE messages expose a new side channel allowing an attacker to use the time between a CBGP message and its corresponding FIZZLE to infer more information than is explicitly revealed by ground-BGP. Note that, given that every CBGP is guaranteed to eventually elicit a corresponding FIZZLE, the information leaked by this side channel is limited to the convergence delay. Third, if a single provider owns the entire network infrastructure in the CIRCA cloud, it can monitor control traffic patterns (despite encryption) to derive more information than is realistically possible in ground-BGP today. We defer further analyses of and defenses against these and other information leakage attacks to future work.

Security vulnerabilities. CIRCA's design allows BGP in the distributed cloud protocol to be drop-in replaced by S-BGP [23] (or other related security extensions like soBGP, BGPSec, etc.) while qualitatively preserving its convergence delay benefits as well as safety, liveness, and route computation equivalence guarantees provided the secured variant of BGP is guaranteed to produce routing outcomes *equivalent* to its unsecured version. CIRCA largely continues to be vulnerable to protocol *manipulation attacks* [47] against which cryptographic mechanisms as in S-BGP alone cannot protect, however known manipulation attacks based on abusing MRAI timers allowing an off-path adversary to potentially permanently disconnect good ASes despite the existence of good policy-compliant paths between them are ineffective in CIRCA because of its avoidance of MRAI timers.

CIRCA's liveness guarantee (§3.3.2) relies on the safety of BGP policies, which potentially allows a rogue AS to change its policies (in a possibly self-defeating manner) simply to stall the CIRCA control plane. Although the rogue AS could mount this attack even in ground-BGP today, an ameliorating factor on the ground is that routers will continue to update their FIBs and forward traffic throughout the never-ending convergence period, potentially over loops or blackholes. In the CIRCA cloud in contrast, any root cause event—necessarily a policy change event (as opposed to a link/node down/up event)—that results in a safety violation will never converge, so ground routers will never receive the updated FIBs corresponding to that particular event. However, there are two alleviating factors in CIRCA. First, ground routers can fall back on ground BGP with CIRCA's support for BGP co-existence for that root event. Second, other root events can proceed unaffected with CIRCA's default "careless" concurrent event processing approach. These factors suggest that CIRCA-enabled BGP will be no worse than BGP in the presence of safety-violating policy attacks.

Non-convergent path exploration can be cleanly limited by augmenting the CIRCA cloud protocol with a TTL (time-to-live) mechanism that aborts (or force-fizzles) path exploration along any path in the directed message tree after a predefined maximum number of hops. A second design choice is to augment the cloud protocol with a TTL that upon expiration causes a cloud router to dispatch its current FIB to its ground avatar, reset the TTL to its maximum value, and resume the (never-ending) path exploration phase for that root event, a design that in effect induces ground routers to jump from one set of potentially inconsistent FIBs to another in a never-ending manner (similar to BGP). Such adaptations will not preserve RCE as defined because RCE is not well defined in the absence of ECC that will not hold under unsafe or non-convergent BGP policies, however the second design choice in practice comes close to emulating BGP behavior in non-convergent scenarios. We defer a more detailed design and analysis of CIRCA with unsafe BGP policies to future work.

Honest-but-curious threat model. CIRCA as described herein trusts each (replicated) cloud provider to provide and maintain the physical infrastructure in a non-curious manner, but its design can be extended to support a honest-but-curious cloud provider. One option is to employ emerging secure computing platforms [34] to prevent the entity controlling the physical machine from snooping on protected customer data within the machine, an approach that does however implicitly involve the manufacturer of the secure computing processor (e.g, Intel with SGX). To prevent a cloud provider controlling the OS on the machine from using the pattern of memory accesses from leaking information, further techniques such as Oblivious RAM [48] will be required. A quirkier alternative is to organize each replicated CIRCA location similar in spirit to a global exchange point with a "bring-your-own-hardware" model in conjunction with physical security mechanisms, a design wherein the CIRCA cloud itself is federated obviating a trusted infrastructure provider.

Security and robustness benefits. Our hypothesis is that the long-term potential benefits of CIRCA are likely to outweigh the impact of new vulnerabilities it introduces. First, CIRCA provides a clean slate enabling early adopters to employ from the get go secure BGP mechanisms that have seen two decades of research and standardization work but little deployment in practice. Second, with willing AS participants, it is intrinsically easier to monitor select portions of control traffic in the CIRCA cloud compared to ground-BGP today in order to detect misbehavior such as protocol manipulation attacks. Third, CIRCA enables new opportunities such as augmenting the CIRCA cloud with mechanisms for "what-if" analysis allowing operators to ascertain whether an action will have the intended effect before performing that action.

6 Conclusions

In this paper, we presented the design, formal analysis, implementation, and prototype-driven evaluation of CIRCA, a logically centralized architecture for interdomain routing control. Although logical centralization of network control is a widely embraced direction in recent years in intradomain networks, attempts to extend that vision to Internet-wide interdomain routing have been limited to on-paper designs or small-scale evaluations. To our knowledge, this is the first work to present a full-fledged design and implementation of an interdomain routing control platform. Our underlying technical contributions include a novel distributed convergence detection algorithm; demonstrating the potential for significantly reducing BGP's tail latencies; formally ensuring route computation equivalence with a broad class of BGP-like protocols, and designing mechanisms for enabling incremental deployment and coexistence with BGP.

References

[1] Michael Alan Chang, Thomas Holterbach, Markus Happe, and Laurent Vanbever. Supercharge me: Boost router convergence with SDN. *ACM SIGCOMM Computer Communication Review*, 45(4):341–342, 2015.

[2] Gilad Asharov, Daniel Demmler, Michael Schapira, Thomas Schneider, Gil Segev, Scott Shenker, and Michael Zohner. Privacy-preserving interdomain routing at internet scale. *Proceedings on Privacy Enhancing Technologies*, 2017(3):147–167, 2017.

[3] Zied Ben Houidi, Mickael Meulle, and Renata Teixeira. Understanding slow BGP routing table transfers. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 350–355, 2009.

[4] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 15–28. USENIX Association, 2005.

[5] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM computer communication review*, 37(4):1–12, 2007.

[6] Marco Chiesa, Daniel Demmler, Marco Canini, Michael Schapira, and Thomas Schneider. Towards securing internet exchange points against curious onlookers. In *Proceedings of the 2016 Applied Networking Research Workshop*, pages 32–34. ACM, 2016.

[7] Marco Chiesa, Daniel Demmler, Marco Canini, Michael Schapira, and Thomas Schneider. Sixpack: Securing internet exchange points against curious onlookers. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 120–133, 2017.

[8] Matthew L Daggitt, Alexander JT Gurney, and Timothy G Griffin. Asynchronous convergence of policy-rich distributed bellman-ford routing protocols. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 103–116, 2018.

[9] Alex Fabrikant, Umar Syed, and Jennifer Rexford. There's something about MRAI: Timing diversity can exponentially worsen BGP convergence. In *2011 Proceedings IEEE INFOCOM*, pages 2975–2983. IEEE, 2011.

[10] Adrian Gämperli, Vasileios Kotronis, and Xenofontas Dimitropoulos. Evaluating the effect of centralization on routing convergence on a hybrid BGP-SDN emulation framework. *ACM SIGCOMM Computer Communication Review*, 44(4):369–370, 2014.

[11] Lixin Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Transactions on Networking (ToN)*, 9(6):733–745, 2001.

[12] Timothy G Griffin and Brian J Premore. An experimental analysis of BGP convergence time. In *Network Protocols, 2001. Ninth International Conference on*, pages 53–61, IEEE, 2001. IEEE, IEEE.

[13] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking (ToN)*, 10(2):232–243, 2002.

[14] Arpit Gupta, Nick Feamster, and Laurent Vanbever. Authorizing network control at software defined internet exchange points. In *Proceedings of the Symposium on SDN Research*, page 16. ACM, 2016.

[15] Arpit Gupta, Robert MacDavid, Rudiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. An industrial-scale software defined internet exchange point. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 1–14, 2016.

[16] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A software defined internet exchange. *ACM SIGCOMM Computer Communication Review*, 44(4):551–562, 2015.

[17] Debayan Gupta, Aaron Segal, Aurojit Panda, Gil Segev, Michael Schapira, Joan Feigenbaum, Jenifer Rexford, and Scott Shenker. A new approach to interdomain routing based on secure multi-party computation. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 37–42, 2012.

[18] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[19] P JAKMA. Revised default values for the BGP 'Minimum Route Advertisement Interval'. https://tools.ietf.org/id/draft-jakma-mrai-00.html, November 2008. [Online; accessed 15-March-2019].

[20] John P John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus routing: The internet as a distributed system. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 351–364, 2008.

[21] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[22] Howard Karloff. On the convergence time of a path-vector protocol. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, page 605–614, USA, 2004. Society for Industrial and Applied Mathematics.

[23] Stephen T. Kent, Charles Lynn, and Karen Seo. Secure border gateway protocol (S-BGP). *IEEE Journal on Selected Areas in Communications*, 18(4):582–592, 2000.

[24] Vasileios Kotronis, Xenofontas Dimitropoulos, and Bernhard Ager. Outsourcing the routing control logic: better internet routing based on SDN principles. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 55–60, 2012.

[25] Vasileios Kotronis, Xenofontas Dimitropoulos, and Bernhard Ager. Outsourcing routing using SDN: The case for a multi-domain routing operating system. *Poster Proc. of ONS*, 2013.

[26] Vasileios Kotronis, Adrian Gämperli, and Xenofontas Dimitropoulos. Routing centralization across domains via SDN: A model and emulation framework for BGP evolution. *Computer Networks*, 92:227–239, 2015.

[27] Vasileios Kotronis, Rowan Klöti, Matthias Rost, Panagiotis Georgopoulos, Bernhard Ager, Stefan Schmid, and Xenofontas Dimitropoulos. Stitching inter-domain paths over IXPs. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[28] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed internet routing convergence. *ACM SIGCOMM Computer Communication Review*, 30(4):175–187, 2000.

[29] Craig Labovitz, Abha Ahuja, Roger Wattenhofer, and Srinivasan Venkatachary. The impact of internet policy and topology on delayed routing convergence. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society*, pages 537–546. IEEE, 2001.

[30] Karthik Lakshminarayanan, Ion Stoica, Scott Shenker, and Jennifer Rexford. *Routing as a Service*. Computer Science Division, University of California Berkeley, 2004.

[31] Anthony Lambert, Marc-Olivier Buob, and Steve Uhlig. Improving internet-wide routing protocols convergence with MRPC timers. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 325–336. ACM, 2009.

[32] Nenad Laskovic and Ljiljana Trajkovic. BGP with an adaptive minimal route advertisement interval. In *2006 IEEE International Performance Computing and Communications Conference*, pages 8–pp. IEEE, 2006.

[33] Changhyun Lee, Keon Jang, and Sue Moon. Reviving delay-based TCP for data centers. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 111–112. ACM, 2012.

[34] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, New York, NY, USA, 2016. Association for Computing Machinery.

[35] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Ricardo Oliveira, Beichuan Zhang, Dan Pei, and Lixia Zhang. Quantifying path exploration in the internet. *IEEE/ACM Transactions on Networking (TON)*, 17(2):445–458, 2009.

[37] Dan Pei, Beichuan Zhang, Dan Massey, and Lixia Zhang. An analysis of path-vector routing protocol convergence algorithms. *Computer Networks*, 50(3):398–421, 2006.

[38] Shahrooz Pouryousef, Lixin Gao, and Arun Venkataramani. Towards Logically Centralized Interdomain Routing, UMass CICS Technical Report, UM-CS-2020-001. https://web.cs.umass.edu/publication/docs/2020/UM-CS-2020-001.pdf, February 2020.

[39] RFC. A border gateway protocol 4 (BGP-4). https://tools.ietf.org/html/rfc4271.

[40] RFC. A border gateway protocol 4 (BGP-4). https://tools.ietf.org/html/rfc4098.

[41] Christian Esteve Rothenberg, Marcelo Ribeiro Nascimento, Marcos Rogerio Salvador, Carlos Nilton Araujo Corrêa, Sidney Cunha de Lucena, and Robert Raszuk. Revisiting routing control platforms with the eyes and muscles of software-defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 13–18. ACM, 2012.

[42] RouteViews. Routeviews. http://www.routeviews.org.

[43] Amit Sahoo, Krishna Kant, and Prasant Mohapatra. Improving BGP convergence delay for large-scale failures. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 323–332. IEEE, 2006.

[44] Pavlos Sermpezis and Xenofontas Dimitropoulos. Inter-domain SDN: Analysing the effects of routing centralization on BGP convergence time. *ACM SIGMETRICS Performance Evaluation Review*, 44(2):30–32, 2016.

[45] J. L. S. Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Transactions on Networking*, 13(5):1160–1173, October 2005.

[46] J. L. S. Sobrinho and T. Griffin. Routing in equilibrium. In *International Symp. on the Mathematical Theory of Networks and Systems - MTNS*, pages –, July 2010.

[47] Yang Song, Arun Venkataramani, and Lixin Gao. Identifying and addressing protocol manipulation attacks in "secure" BGP. In *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013, 8-11 July, 2013, Philadelphia, Pennsylvania, USA*, pages 550–559, 2013.

[48] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. *J. ACM*, 65(4), April 2018.

[49] Martin Suchara, Alex Fabrikant, and Jennifer Rexford. BGP safety with spurious updates. In *INFOCOM*, pages 2966–2974, 2011.

[50] Wei Sun, Zhuoqing Morley Mao, and Kang G Shin. Differentiated BGP update processing for improved routing convergence. In *Proceedings of the 2006 IEEE International Conference on Network Protocols*, pages 280–289. IEEE, 2006.

[51] Feng Wang and Lixin Gao. A backup route aware routing protocol-fast recovery from transient routing failures. In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pages 2333–2341. IEEE, 2008.

[52] Hong Yan, David A Maltz, TS Eugene Ng, Hemant Gogineni, Hui Zhang, and Zheng Cai. Tesseract: A 4d network control plane. In *NSDI*, volume 7, pages 27–27, 2007.

[53] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 432–445. ACM, 2017.

[54] Ziyao Zhang, Liang Ma, Kin K Leung, Franck Le, Sastry Kompella, and Leandros Tassiulas. How better is distributed SDN? an analytical approach. *arXiv preprint arXiv:1712.04161*, 2017.

[55] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. ECN or delay: Lessons learnt from analysis of DCQCN and TIMELY. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 313–327. ACM, 2016.

A Algorithm 1: Safety and liveness proofs

In this section, we formally prove the safety and liveness properties satisfied by Algorithm 1. The proofs rely on a construction called the *directed message graph* produced by any execution instance of Algorithm 1 and defined as follows.

Definition 3. DIRECTED MESSAGE GRAPH: *The directed message graph of an instance of Algorithm 1 is a directed graph $G = (\mathcal{V}, \mathcal{E})$ such that each vertex in $\mathcal{V}$ denotes a sent message (either the original* GRC *or a* CBGP *message) and $\mathcal{E}$ has a directed edge from a message m_1 to another message m_2 if m_1 caused m_2, i.e., m_2 was sent in lines 14 or 26 respectively in response to the receipt of m_1 in lines 16 or 1.*

We say that $m_1 \to m_2$ (or m_1 *happened before* m_2) if there is a directed path from message m_1 to m_2 in the graph.

Lemma A.1. *The directed message graph produced by any execution of Algorithm 1 is a directed tree.*

Proof. Acyclicity follows from the observation that each message sent by Algorithm 1 is unique, as identified by the unique two-tuple timestamp assigned in lines 11 or 23, thus the directed message graph is a directed acyclic graph (DAG). That DAG is also a tree because each message is caused by at most one other message: a CBGP message is caused by either a unique CBGP message (line 13) or the original GRC (line 25) and the original GRC being the root of the DAG is not caused by any other message. □

In the proofs below, we will use *tree*(m) denote the subtree rooted at m in the directed message tree produced by an instance of Algorithm 1 (which is well-defined because of Lemma A.1 just above).

A.1 Proof of safety

We introduce the following lemmas in order to prove safety (Theorem 3.1).

Lemma A.2. *If a router receives a* FIZZLE *for a* CBGP *message m_1 that it sent, then tree(m_1) is finite.*

Proof. Suppose the execution of Algorithm 1 started at time 0 and the router in the lemma's premise received the FIZZLE for m_1 at some (finite) time t_1. Further assume that the time since a router sends a FIZZLE until the corresponding neighboring router receives it is lower bounded by a constant time $c > 0$ (as would be the case in any real implementation of Algorithm 1). To show that *tree*(m_1) is finite, we show that every path rooted at m_1 is finite as follows.

Consider the set *caused*(m_1) of the immediate children of m_1, which is the set of messages sent by the recipient of m_1 in either the `for` loop on either line 10 (if m_1 were a GRC message) or line 22 (if m_1 were a CBGP message). By inspection of Algorithm 1, a FIZZLE is sent only at two places, line 20 and line 35. In the former case (line 20), the causal message m_1 did not spawn any further child messages at the recipient of m_1, so *caused*(m_1) is empty implying that *tree*(m_1) is of unit size.

In the latter case (line 35), the recipient of m_1 must have previously remembered the set of caused messages *caused*(m_1) in its local map (the *sent*$[E][ts(m_1)]$ map in lines 12 or 24) and subsequently sent a FIZZLE for m_1 back to its sender when the *sent* map got emptied at the recipient (line 34), i.e., the recipient of m_1 received a FIZZLE for every message in *caused*(m_1). By assumption, the recipient of m_1 must have received a FIZZLE for every message in *caused*(m_1) by time no later than $t - c$.

Let *caused*$^k(m_1)$ recursively denote the set of messages caused by *caused*$^{k-1}(m_1)$ for $k > 1$. By repeating the above argument, all messages in *caused*$^k(m_1)$ must have received a corresponding FIZZLE from their respective recipients by time $t - kc$. By assumption, the algorithm began execution at time 0, so the depth of *tree*(m_1) is at most t/c.

□

Lemma A.3. *A router receives a* FIZZLE *for a* CBGP *message m_1 it sent only if for every message m_2 (sent by any router) such that $m_1 \to m_2$, the sender of m_2 had previously received a matching* FIZZLE *from the recipient of m_2.*

Proof. The proof is by induction on the depth of the subtree *tree*(m_1) rooted at m_1, which by Lemma A.2 is finite.

Consider a topological sort of *tree*(m_1) with the root m_1 at level 0 and all the messages in *caused*$^k(m_1)$ at level $k > 0$. Let the depth or maximum level be d (that is well defined because of Lemma A.2). The lemma's claim is trivially true for a level d message as it did not cause any further messages. Suppose that the claim is true for messages at all levels in $[i, d]$ (both inclusive) for some $1 < i < d$. Consider a non-leaf message m at level $i - 1$ sent by a router for some root event E. By line 35, the recipient router can issue a FIZZLE for m only if its *sent*$[E]$ is empty, i.e., only if it has received a matching FIZZLE for every message in *caused*(m), which completes the inductive step for level i, proving the lemma. □

Lemma A.4. *When converged(E) is set to true at the root router, sent$[E]$ is empty at every router.*

Proof. There are two cases to consider.

Case 1: *converged(E) is set upon receipt of a* GRC *message*

In this case, the root router will terminate before sending any CBGP message. No router will send any further CBGP messages because a CBGP message can only be sent in response to the receipt of a CBGP or GRC message.

Case 2: *converged(E) is set upon receipt of a* FIZZLE.

By inspection of code, this case can happen only if the router receiving the FIZZLE is the router that received the root cause GRC message. *converged*(E) is set at the root router only when its *sent*$[E]$ map is empty (line 7). By inspection (line 31), an entry in *sent*$[E]$ is removed *only if* the matching FIZZLE is received. Thus, the root router must have received a matching FIZZLE for each message it sent including all the level 1 messages immediately caused by the root GRC message. By Lemma A.3, every sent message for E (at any router) must have received a matching FIZZLE. By line 31, an entry in *sent*$[E]$ is removed *if* the matching FIZZLE is received. So, if every sent message for E has received a matching FIZZLE, *sent*$[E]$ must be empty at all routers, proving the claim. □

We complete the proof of safety as follows.

Theorem 3.1. SAFETY: If *converged(E)* is true at the root cloud router that received the GRC, then no further CBGP message for E will be received by any cloud router.

Proof. By Lemma A.4 above, *sent*$[E]$ is empty at every router at this point, so there is no in-propagation CBGP message for E as every sent message at any router has already received the corresponding FIZZLE. The root router that received the GRC message terminates immediately after setting *converged*(E), so it will not send any further CBGP messages for E. A CBGP message can be sent by a non-root router (line 26) only upon receiving an in-propagation CBGP message, so no non-root router will send any further CBGP messages either. □

Technical note. If we assume safe BGP policies, the onerousness in Lemma A.2 would be unnecessary as the directed message tree would be bounded by definition. However, the proof of CIRCA's safety (unlike the proof of its liveness immediately below) does not require the safety of the underlying BGP policies, so we chose the more onerous formal route to make that independence clear.

A.2 Proof of liveness

The liveness proofs below implicitly assume that routers in the CIRCA cloud either do not fail or employ a persistent write-ahead log to record all local state changes before sending messages based on those changes. (Unlike liveness, neither failures nor the availability of a persistent log impacts safety.)

Lemma A.5. *If BGP policies are safe, every sent* CBGP *message eventually receives a matching* FIZZLE *if all cloud routers are available for a sufficiently long duration.*

Proof. The proof of this claim is by induction and is similar in spirit to Lemma A.3. The assumed safety of BGP policies by definition bounds the size of the directed message tree. Consider a topological sort as before partitioning the tree into levels where level k nodes are the messages in $caused^k(\text{GRC})$ where we have used GRC as a shorthand for the original GRC message that initiated Algorithm 1 at the root router. Let d denote the depth of the tree.

The claim is true for level d messages because of lines 19–26: a router receiving a CBGP message must either cause a new CBGP (the else block starting line 21) or issue a FIZZLE for the received CBGP (line 20). Level d messages do not cause any further CBGP messages, so routers receiving them will issue a corresponding FIZZLE.

The inductive step is as follows. Suppose the claim holds for levels i to d for $1 < i < d$. Consider a level $i-1$ message m with timestamp ts received by a router that causes it to send the set $caused(m)$ of level i messages. By the inductive assumption, each level i message m_1 eventually receives a matching FIZZLE, and each such FIZZLE removes the corresponding entry from $sent[E][ts_1]$ at the router that received m_1 where ts_1 is the timestamp of m_1. Thus, when the recipient of m has received matching FIZZLEs for all level i messages caused by the incoming level $i-1$ message m, its $sent[E][ts]$ map becomes empty triggering the matching FIZZLE for m, which completes the inductive step, proving the lemma. □

Theorem 3.2. LIVENESS: Algorithm 1 eventually terminates if the underlying BGP policies are safe[2] and all cloud routers are available for a sufficiently long duration.

Proof. To prove that Algorithm 1 terminates, i.e., *converged*(E) is set at the root router, we consider two cases, the first of which as in the safety proof is trivial.

Case 1: *No* CBGP *messages are sent.*

In this case, the algorithm trivially terminates in line 7.

Case 1: CBGP *messages are sent.*

The proof of the theorem follows by observing that, by Lemma A.5, the root router must eventually receive a matching FIZZLE for all (level 1) CBGP messages caused by the root GRC (in line 26). At this point, its *sent*$[E]$ map must be empty for the following reason: Lemma A.3 implies that any level 2 or higher CBGP message sent by any router must have already fizzled, so any level 2 or higher CBGP sent by the root router must have also already fizzled and consequently already removed from its *sent*$[E]$ map. By line 32, an empty *sent*$[E]$ map causes Algorithm 1 to terminate. □

Note that although the proofs above did not explicitly invoke the "*all cloud routers are available for a sufficiently long duration*" assumption, they implicitly relied on that in conjunction with a persistent write-ahead log in all claims with the word "*eventually*" in the proofs above.

[2]A reminder that "safe" here means that a stable route configuration exists [13] and is unrelated to *safety* in Theorem 3.1

A.3 Proof of efficiency

Algorithm 1 omitted a formal description of the third *dissemination* phase for a concise exposition of the key safety and liveness properties. For completeness, we codify the dissemination phase in event-action format as well. The root router that received the GRC message for event E, instead of exiting when $converged(E)$ becomes true (on line 33), initiates the dissemination phase by sending to itself the COMMIT message $\langle$COMMIT$, E, ts\rangle$ where ts is the timestamp of the original GRC message as assigned on line 9 in Algorithm 1. The event handler for a received commit message is below.

Dissemination phase of Algorithm 1

```
1: event RECV(⟨COMMIT, $E, ts$⟩, $N$)   ▷ upon receipt of a commit
   message from cloud router N at cloud router $v(R)$
2:     $p \leftarrow$ all affected prefixes in $fizzled[E][ts]$
3:     send(⟨FBGP, $E, p, FIB(p)$⟩, R)   ▷ southbound FIB dispatch
4:     for each $[N_1, p_1, ts_1]$ : $fizzled[E][ts]$ do
5:         send(⟨COMMIT, $E, ts_1$⟩, $N_1$)
6:     $fizzled[E][ts] \leftarrow \emptyset$
7:     if $fizzled[E] = \emptyset$ then exit
```

THEOREM 3.3. EFFICIENCY: *The root router (Any router) in Algorithm 1 detects convergence within at most 2Δ (3Δ) time and $2M$ ($3M$) messages where Δ and M are respectively the actual convergence delay and number of messages incurred by the underlying distributed route computation protocol.*

Proof. The convergence indicating COMMIT messages disseminated in Algorithm 1's third phase by design follow the exact same paths as the CBGP messages in the directed message tree in the first (exploration) phase. This claim follows from the observation that when $converged(E)$ is set at the root router, Lemma A.4 implies that every entry ever previously added to the *sent*[E] map (in lines 12 or 24) has already been removed, and by inspection of lines 31 and 30, every entry removed from the $sent[E]$ map is added to the $fizzled[E]$ map.

Assuming safety of the underlying BGP policies, every CBGP message eventually begets a matching FIZZLE in the reverse direction in the second (back-propagation) phase and a matching COMMIT in the same direction in the third (dissemination) phase, thus the number of messages in each phase is identical. The "*actual convergence delay and number of messages incurred by the underlying distributed route computation protocol*" are defined as those of the first phase alone. The root router detects convergence at the end of the second phase when $converged(E)$ becomes true. Each router detects convergence when its $fizzled[E]$ map gets emptied, which necessarily happens for every router in the third phase by an inductive argument similar to that used in Lemma A.5. □

Technical notes. The efficiency proof conservatively assumes that the second and third phases incur at most as much delay as the first phase. In a practical implementation, this assumption is more than true in that the first phase is likely to incur much more delay than the other two phases. The reason is that the first phase CBGP messages may carry a large number (hundreds or thousands) of prefixes resulting in large transfer sizes at each hop but the second and third phase messages are small control messages as they just need to convey the root cause event identifier and timestamp.

The dissemination phase as strictly codified above may result in any given router dispatching southward portions of the modified FIB entries for a single root cause event in multiple bursts (via line 3) in the third phase. It is straightforward to modify the protocol so as to either dispatch the entire set of modified FIB entries as a single message just before exit (on line 7) instead or send a termination marker southward just before exiting so that the ground incarnate can apply all modified FIB entries atomically. This modification may reduce transient inconsistencies in the ground forwarding plane, especially if cloud routers intermittently fail and pre-computed backup routing options are available to ground routers.

B Route computation equivalence

THEOREM 3.4. ROUTE COMPUTATION EQUIVALENCE: *If for a sufficiently long period—(i) all ground routers can reach a* CIRCA *cloud datacenter and vice versa; and (ii) all cloud routers are available and can communicate in a timely manner—a pure* CIRCA *system ensures Route Computation Equivalence with any distributed route computation function that satisfies Eventually Consistent Convergence.*

Proof. Let $\mathbf{D}(S)$ denote any distributed route computation function. Suppose the initial network state is S_0 and a sequence of root cause events $[e_1, \ldots, e_n]$ (but no further events) occurs. Consider the following mutually exclusive and exhaustive set of event sequences: $\{[e_{11}, e_{12}, \ldots], [e_{21}, e_{22}, \ldots], \ldots, [e_{m1}, e_{m2}, \ldots]\}$ where e_{ij} denotes an event at router i with (local) sequence number j and m denotes the total number of routers, i.e., each sequence in this set is an ordered subset of events in the original event sequence all of which were detected by the same router in that local order.

The theorem follows from the following claims: (1) CIRCA eventually processes all root events; (2) CIRCA processes root events in an order consistent with the local order at ground routers; (3) processing the root events in any global order consistent with local orders results in the same final routing outcomes. The first two claims are straightforward: the first follows from the assumption of sufficiently long periods of cloud-ground reachability and cloud router availability; the second follows from line II.1 in the north-south protocol.

The last of the above claims is the non-intuitive one and needs a proof. The proof surprisingly follows from the assumption that the ground routing protocol ensures Eventually Consistent Convergence. Consider any reordering of the $\mathcal{F} = [f_1, \ldots, f_n]$ of the original event sequence $\mathcal{E} = [e_1, \ldots, e_n]$ that preserves local order. We claim that $S_0|\mathcal{F} = S_0|\mathcal{E}$, i.e., the network arrives at the same final state given any local-order-preserving reordering[3] of an event sequence.

To see why, suppose events only consisted of link up and link down events for a given link. A link event will be detected and reported by one or both of the incident ground routers, say A and B. No matter how many times the link goes up or down in a given sequence of events, in any local-order-preserving reordering of that sequence, both A and B will agree on the final state of the link. Note that the claim is not true in general for a non-local-order-preserving reordering, for example, if link $A-B$ went down and then came back up, but the last event was reported by A and reported as a link down event, the final state of the network (and by consequence routing outcomes) will be different.

The complete proof follows from showing the third claim, namely that any *sequentially consistent* reordering of events produces the same final outcome, in a manner similar to above for other root cause events including node up/down events, link cost changes, as well as combinations of such events. □

We conjecture that RCE given ECC as above holds for more general policy or other configuration changes at routers. It is straightforward to see that any set of configuration changes across distributed routers will preserve RCE if the final state is the same given any sequentially consistent reordering of those changes. However, it is unclear to us if this property always holds or to what extent it holds in practice. For example, if the value of a router configuration parameter is determined based on the value of another parameter at a different router, the final network configuration state may depend on the precise total order of all events, i.e., it is sensitive to different reorderings even if they are sequentially consistent.

B.1 Practical considerations for RCE

The high-level CIRCA design may require or benefit from some adaptations in practice as summarized below in order to preserve route computation equivalence. A more detailed investigation of these is deferred to future work.

IGP-awareness: A root event such as a link cost change or, more generally, any intradomain root event that potentially affects interdomain routes at any router needs to be conveyed somehow to that router. In an intradomain routing protocol such as OSPF, link-state advertisements (LSA) accomplish that purpose. There are two natural design alternatives in CIRCA to accomplish the same: (1) piggypack the root cause event label in LSAs similar to CIRCA's CBGP messages; (2) use iBGP to disseminate the root event network-wide within the domain. The latter approach has the benefit of being largely decoupled from the intradomain routing protocol and works well for intradomain routing protocols that are *global* by design like link-state routing, but not so well for *decentralized* protocols like distance-vector routing wherein a router by design does not know the final outcome of the route computation until the decentralized computation completes. In practice, "full-mesh-iBGP-as-LSA" suffices to maintain route computation equivalence for any shortest path routing protocol or, more generally, any intradomain routing protocol that lends itself to a global implementation, i.e., one where an individual router can immediately compute its final intradomain routing outcome for any given change to global (intradomain) network state.

Root cause event grammar: Although the variable-length description $\langle etype, eval \rangle$ (refer §3.1.1 and Figure 4) is in principle sufficiently general to allow CIRCA to represent arbitrary changes to router configuration state, and parsing it is a purely intradomain concern, there may be value in minimally standardizing this representation across different router vendors for the sake of a reusable implementation. For example, the unix utility `diff` is a naive way to represent changes to router configuration files in a platform-agnostic manner. A more systematic vendor-specific or -neutral grammar for representing root cause events is conceivable.

Domain consolidation: Consolidating route computation for routers in the same domain by employing a single well-provisioned router server (similar in spirit to RCP [4]) instead of one-one mapping ground routers to virtual routers is likely to improve resource efficiency and thereby reduce the provisioning cost of route servers. The consolidated approach blurs the distinction between the abstract "single-router-AS" model and ASes in practice, arguably making it easier for operators to monitor and understand routing dynamics within their networks. Maintaining route computation equivalence requires that the consolidated server is guaranteed to computes a routing outcome that could have been computed by the correspondning ground protocol.

[3]Or a *sequentially consistent* ordering in distributed computing parlance.

XRD: Scalable Messaging System with Cryptographic Privacy

Albert Kwon — *MIT*
David Lu — *MIT PRIMES*
Srinivas Devadas — *MIT*

Abstract

Even as end-to-end encrypted communication becomes more popular, private messaging remains a challenging problem due to metadata leakages, such as who is communicating with whom. Most existing systems that hide communication metadata either (1) do not scale easily, (2) incur significant overheads, or (3) provide weaker guarantees than cryptographic privacy, such as differential privacy or heuristic privacy. This paper presents XRD (short for Crossroads), a metadata private messaging system that provides cryptographic privacy, while scaling easily to support more users by adding more servers. At a high level, XRD uses multiple mix networks in parallel with several techniques, including a novel technique we call aggregate hybrid shuffle. As a result, XRD can support 2 million users with 228 seconds of latency with 100 servers. This is 13.3× and 4× faster than Atom and Pung, respectively, which are prior scalable messaging systems with cryptographic privacy.

1 Introduction

Many Internet users today have turned to end-to-end encrypted communication like TLS [18] and Signal [40], to protect the *content* of their communication in the face of widespread surveillance. While these techniques are starting to see wide adoption, they unfortunately do not protect the *metadata* of communication, such as the timing, the size, and the identities of the end-points. In scenarios where the metadata are sensitive (e.g., a government officer talking with a journalist for whistleblowing), encryption alone is not sufficient to protect users' privacy.

Given its importance, there is a rich history of works that aim to hide the communication metadata, starting with mix networks (mix-nets) [10] and dining-cryptographers networks (DC-Nets) [11] in the 80s. Both works provide formal privacy guarantees against global adversaries, which has inspired many systems with strong security guarantees [14, 55, 35, 53]. However, mix-nets and DC-nets require the users' messages to be processed by either centralized servers or every user in the system, making them difficult to scale to millions of users. Systems that build on them typically inherit the scalability limitation as well, with overheads increasing (often superlinearly) with the number of users or servers [14, 55, 35, 53]. For private communication systems, however, supporting a large user base is imperative to providing strong security; as aptly stated by prior works, "anonymity loves company" [20, 49]. Intuitively, the adversary's goal of learning information about a user naturally becomes harder as the number of users increases.

As such, many recent messaging systems have been targeting scalability as well as formal security guarantees. Systems like Stadium [52] and Karaoke [37], for instance, use differential privacy [22] to bound the information leakage on the metadata. Though this has allowed the systems to scale to more users with better performance, both systems leak a small bounded amount of metadata for each message, and thus have a notion of "privacy budget". A user in these systems then spends a small amount of privacy budget every time she sends a sensitive message, and eventually is not guaranteed strong privacy. Users with high volumes of communication could quickly exhaust this budget, and there is no clear mechanism to increase the privacy budget once it runs out. Scalable systems that provide stronger cryptographic privacy like Atom [34] or Pung [5], on the other hand, do not have such a privacy budget. However, they rely heavily on expensive cryptographic primitives such as public key encryption and private information retrieval [3]. As a result, they suffer from high latency, in the order of ten minutes or longer for a few million users, which impedes their adoption.

This paper presents a point-to-point metadata private messaging system called XRD that aims to marry the best aspects of prior systems. Similar to several recent works [5, 34, 52], XRD scales with the number of servers. At the same time, the system cryptographically hides all communication metadata from an adversary who controls the entire network, a constant fraction of the servers, and any number of users. Consequently, it can support virtually unlimited amount of communication without leaking privacy against such an adversary. Moreover, XRD only uses cryptographic primitives that are significantly faster than the ones used by prior works, and can thus provide lower latency and higher throughput than prior systems with cryptographic security.

A XRD deployment consists of many servers. These servers are organized into many small chains, each of which acts as a local mix-net. Before any communication, each user creates a *mailbox* that is uniquely associated with her, akin to an e-mail address. In order for two users Alice and Bob to have a conversation in the system, they first pick a number of chains using a specific algorithm that XRD provides. The algorithm guarantees that *every* pair of users intersects at one of the chains. Then, Alice and Bob send messages addressed to their own mailboxes to all chosen chains, except to the chain where their choices of chains align, where they send their messages for each other. Once all users submit their messages, each chain shuffles and decrypts the messages, and forwards the shuffled messages to the appropriate mailboxes. Intuitively, XRD protects the communication metadata because (1) every pair of users is guaranteed to meet at a chain which makes it equally likely for any pair of users to be communicating, and (2) the mix-net chains hide whether a user sent a message to another user or herself.

One of the main challenges of XRD is addressing active attacks by malicious servers, where they tamper with some of the users' messages. This challenge is not new to our system, and several prior works have employed expensive cryptographic primitives like verifiable shuffle [14, 55, 35, 52, 34] or incurred significant bandwidth overheads [34] to prevent such attacks. In XRD, we instead propose a new technique called *aggregate hybrid shuffle* that can verify the correctness of shuffling more efficiently than traditional techniques.

XRD has two significant drawbacks compared to prior systems. First, with N servers, each user must send $O(\sqrt{N})$ messages in order to ensure that every pair of users intersects. Second, because each user sends $O(\sqrt{N})$ messages, the workload of each XRD server is $O(M/\sqrt{N})$ for M users, rather than $O(M/N)$ like many prior scalable messaging systems [34, 52, 37]. Thus, prior systems could outperform XRD in deployment scenarios with large numbers of servers and users, since the cost of adding a single user is higher and adding servers is not as beneficial in XRD.

Nevertheless, our evaluation suggests that XRD outperforms prior systems with cryptographic guarantees if there are less than a few thousand servers in the network. XRD can handle 2 million users (comparable to the number of daily Tor users [1]) in 228 seconds with 100 servers. For Atom [34] and Pung [5, 4], two prior scalable messaging systems with cryptographic privacy, it would take over 50 minutes and 15 minutes, respectively. (These systems, however, can defend against stronger adversaries, as we detail in §2 and §7.) Moreover, the performance gap grows with more users, and we estimate that Atom and Pung require at least 1,000 servers in the network to achieve comparable latency with 2 million or more users. While promising, we find that XRD is not as fast as systems with weaker security guarantees: Stadium [52] and Karaoke [37], for example, would be 3× and 23× faster than XRD, respectively, in the same deployment scenario. In terms of user costs, we estimate that 40 Kbps of bandwidth is sufficient for users in a XRD network with 2,000 servers, and the bandwidth requirement scales down to 1 Kbps with 100 servers.

In summary, we make the following contributions:

- Design and analyze XRD, a metadata private messaging system that can scale by distributing the workload across many servers while providing cryptographic privacy.
- Design a technique called aggregate hybrid shuffle that can efficiently protect users' privacy under active attacks.
- Implement and evaluate a prototype of XRD on a network of commodity servers, and show that XRD outperforms existing cryptographically secure designs.

2 Related work

In this section, we discuss related work by categorizing the prior systems primarily by their privacy properties, and also discuss the scalability and performance of each system.

Systems with cryptographic privacy. Mix-nets [10] and DC-Nets [11] are the earliest examples of works that provide cryptographic (or even information theoretic) privacy guarantees against global adversaries. Unfortunately, they have two major issues. First, they are weak against active attackers: adversaries can deanonymize users in mix-nets by tampering with messages, and can anonymously deny service in DC-Nets. Second, they do not scale to large numbers of users because all messages must be processed by either a small number of servers or every user in the system. Many systems that improved on the security of these systems against active attacks [14, 55, 35, 53] suffer from similar scalability bottlenecks. Riposte [13], a system that uses "private information storage" to provide anonymous broadcast, also requires all servers to handle a number of messages proportional to the number of users, and thus faces similar scalability issues.

A recent system Atom [34] targets both scalability and strong anonymity. Specifically, Atom can scale *horizontally*, allowing it to scale to larger numbers of users simply by adding more servers to the network. At the same time, it provides sender anonymity [46] (i.e., no one, including the recipients, learns who sent which message) against an adversary that can compromise any fraction of the servers and users. However, Atom employs expensive cryptography, and requires the message to be routed through hundreds of servers in series. Thus, Atom incurs high latency, in the order of tens of minutes for a few million users.

Pung [5, 4] is a system that aims to provide metadata private messaging between honest users with cryptographic privacy. This is a weaker notion of privacy than that of Atom, as the recipients (who are assumed to be honest) learn the senders of the messages. However, unlike most prior works, Pung can provide private communication even if *all servers* are malicious by using a cryptographic primitive called computational private information retrieval (CPIR) [12, 3]. Its powerful threat model comes unfortunately at the cost of performance: Though Pung scales horizontally, the amount of work required per user is proportional to the total number of users, resulting in the total work growing superlinearly with the number of users. Moreover, PIR is computationally expensive, resulting in throughput of only a few hundred or thousand messages per minute per server.

Systems with differential privacy. Vuvuzela [53] and its horizontally scalable siblings Stadium [52] and Karaoke [37] aim to provide differentially private (rather than cryptographically private) messaging. At a high level, they hide the communication patterns of honest users by inserting dummy messages that are indistinguishable from real messages, and reason carefully about how much information is leaked at each round. They then set the system parameters such that they could support a number of sensitive messages; for instance, Stadium and Karaoke target 10^4 and 10^6 messages, respectively. Up to that number of messages, the systems allow users to provide a plausible cover story to "deny" their actual actions. Specifically, the system ensures that the probability of Alice conversing with Bob from the adversary's perspective is within e^ε (typically, $e^\varepsilon \in [3,10]$) of the probability of Alice conversing with any other user with only a small failure probability δ (typically, $\delta = 0.0001$). This paradigm shift has allowed the systems to support larger numbers of users with lower latency than prior works.

Unfortunately, systems with differential privacy suffer from two drawbacks. First, the probability gap between two events may be sufficient for strong adversaries to act on. For instance, if Alice is ten times as likely to talk to Bob than Charlie, the adversary may act assuming that Alice is talking to Bob, despite the plausible deniability. Second, there is a "privacy budget" (e.g., 10^4 to 10^6 messages), meaning that a user can deny a limited number of messages with strong guarantees. Moreover, for best possible security, users must constantly send messages, and deny every message. For instance, Alice may admit that she is not in any conversation (thinking this information is not sensitive), but this could have unintended consequences on the privacy of another user who uses the cover story that she is talking with Alice. The budget could then run out quickly if users want the strongest privacy possible: If a user sends a message every minute, she would run out of her budget in a few days or years with 10^4 to 10^6 messages. Although the privacy guarantee weakens gradually after the privacy budget is exhausted, it is unclear how to raise the privacy levels once they have been lowered.

These shortcomings can particularly affect journalists and their sources. Many journalists mention the importance of long-term relationships with their sources for their journalistic process [41, 43], and the difficulty of maintaining private relationships with them. In a differentially private system, if the journalist and the source are ten times as likely to be talking as two other users, then the adversary might simply assume the journalist and the source's relationship. Furthermore, these relationships often last many years [43], which could cause the privacy budget to run out. This may put the journalist and the source in jeopardy.

Scalable systems with other privacy guarantees. The only private communication system in wide-deployment today is Tor [21]. Tor currently supports over 2 million daily users using over 6,000 servers [1], and can scale to more users easily by adding more servers. However, Tor does not provide privacy against an adversary that monitors significant portions of the network, and is susceptible to traffic analysis attacks [19, 30]. Its privacy guarantee weakens further if the adversary can control some servers, and if the adversary launches active attacks [29]. Similar to Tor, most free-route mix-nets [44, 24, 49, 15, 39] (distributed mix-nets where each messages is routed through a small subset of servers) cannot provide strong privacy against powerful adversaries due to traffic analysis and active attacks.

Loopix [47] is a recent iteration on free-route mix-nets, and can provide fast asynchronous messaging. To do so, each user interacts with a semi-trusted server (called "provider" in the paper), and routes her messages through a small number of servers (e.g., 3 servers). Each server inserts small amounts of random delays before routing the messages. Loopix then reasons about privacy using entropy. Unfortunately, the privacy guarantee of Loopix weakens quickly as the adversary compromises more servers. Moreover, Loopix requires the recipients to trust the provider to protect themselves.

3 System model and goals

XRD aims to achieve the best of all worlds by providing cryptographic metadata privacy while scaling horizontally without relying on expensive cryptographic primitives. In this section, we present our threat model and system goals.

3.1 Threat model and assumptions

A deployment of XRD would consist of hundreds to thousands of servers and a large number of users, in the order of millions. Similar to several prior works on distributed private communication systems [34, 52], XRD assumes an adversary that can monitor the entire network, control a fraction f of the servers, and control up to all but two honest users. We assume, however, that there exists a public key infrastructure that can be used to securely share public keys of online servers and users with all participants at any given time. These keys, for example, could be maintained by key transparency schemes [36, 42, 51].

XRD does not hide the fact that users are using XRD. Thus, for best possible security, users should stay online to avoid intersection attacks [31, 16]. XRD also does not protect against large scale denial-of-service (DoS) attacks. It can, however, recover from a small number of benign server failures or disruptions from malicious users. In addition, XRD provides privacy even under DoS, server churn, and user churn (e.g., Alice goes offline unexpectedly without her conversation partner knowing). We discuss the availability properties further in §5 and §7.3.

Finally, XRD assumes that the users can agree to start talking at a certain time out-of-band. This could be done, for example, via two users exchanging this information offline, or by using systems like Alpenhorn [38] that can initiate conversations privately.

Cryptographic primitives. XRD assumes existence of a group of prime order p with a generator g in which discrete log is hard and the decisional Diffie-Hellman assumption holds. We will write $\mathsf{DH}(g^a, b) = g^{ab}$ to denote Diffie-Hellman key exchange. In addition, XRD makes use of authenticated encryption.

Authenticated encryption [6]: XRD relies on an authenticated encryption scheme for confidentiality and integrity, which consists of the following algorithms:

- $c \leftarrow \mathsf{AEnc}(\mathsf{s}, \mathsf{nonce}, m)$. Encrypt message m and authenticate the ciphertext c using a symmetric key s and a nonce nonce. Typically, s is used to derive two other keys for encryption and authentication.
- $(b, m) \leftarrow \mathsf{ADec}(\mathsf{s}, \mathsf{nonce}, c)$. Check the integrity of and decrypt ciphertext c using the key s and a nonce nonce. If the check fails, then $b = 0$ and $m = \perp$. Otherwise, $b = 1$ and m is the underlying plaintext.

In general, the adversary cannot generate a correctly authenticated ciphertext without knowing the secret key used for ADec. We use Encrypt-then-HMAC for authenticated encryption, which has the additional property that the ciphertext serves as a commitment to the underlying plaintext with the secret key being the opening to the commitment [28].

3.2 Goals

XRD has three main goals.

Correctness. Informally, the system is correct if every honest user successfully communicates with her conversation partner after a successful execution of the system protocol.

Privacy. Similar to prior messaging systems [53, 5, 52, 37], XRD aims to provide relationship unobservability [46], meaning that the adversary cannot learn anything about the communication between two honest users. Informally, consider any honest users Alice, Bob, and Charlie. The system provides privacy if the adversary cannot distinguish whether Alice is communicating with Bob, Charlie, or neither. XRD only guarantees this property among the honest users, as malicious conversation partners can trivially learn the metadata of their communication. We provide a more formal definition in Appendix B. (This is a weaker privacy goal than that of Atom [34], which aims for sender anonymity.)

Scalability. Similar to prior work [34], we require that the system can handle more users with more servers. If the number of messages processed by a server is $C(M,N)$ for M users and N servers, we require that $C(M,N) \to 0$ as $N \to \infty$. $C(M,N)$ should approach zero polynomially in N so that adding a server introduces significant performance benefits.

4 XRD overview

Figure 1 presents the overview of a XRD network. At a high level, XRD consists of three different entities: users, mix servers, and mailbox servers. Every user in XRD has a unique *mailbox* associated with her, similar to an e-mail address. The mailbox servers maintain the mailboxes, and are only trusted for availability and not privacy.

To set up the network, XRD organizes the mix servers into many chains of servers such that there exists at least one honest server in each chain with overwhelming probability (i.e., an anytrust group [55]). Communication in XRD is carried out in discrete rounds. In each round, each user selects a fixed set of ℓ chains, where the set is determined by the user's public key. She then sends a fixed size message to each of the selected chains. (If the message is too small or large, then the user pads the message or breaks it into multiple pieces.) Each message contains a destination mailbox, and is onion-encrypted for all servers in the chain.

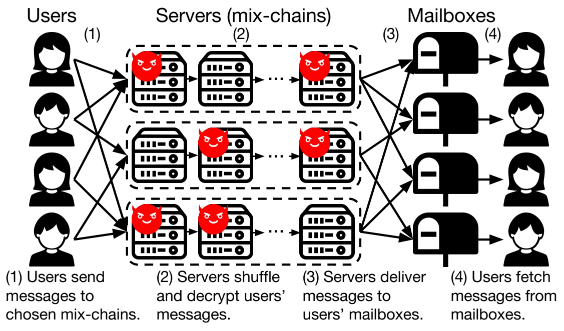

Users
Servers (mix-chains)
Mailboxes
(1)
(2)
(3)
(4)
(1) Users send messages to chosen mix-chains.
(2) Servers shuffle and decrypt users' messages.
(3) Servers deliver messages to users' mailboxes.
(4) Users fetch messages from mailboxes.

Figure 1: Overview of XRD operation.

Once all users submit their messages, each chain acts as a local mix-net [10], decrypting and shuffling messages. During shuffling, each server also generates a short proof that allows other servers to check that it behaved correctly. If the proof does not verify, then the servers can identify who misbehaved. If all verification succeeds, then the last server in each chain forwards the messages to the appropriate mailbox servers. (The protocol for proving and verifying the shuffle is described in §6.) Finally, the mailbox servers put the messages into the appropriate mailboxes, and each user downloads all messages in her mailbox at the end of a round.

The correctness and security of XRD is in large part due to how each user selects the chains. As we will see in §5, the users are required to follow a specific algorithm to select the chains. The algorithm guarantees that *every* pair of users have at least one chain in common and the choices of the chains are publicly computable. For example, every user selecting the same chain will achieve this property, and thus correctness and security. In XRD, we achieve this property while distributing the load evenly.

Let us now consider two scenarios: (1) a user Alice is not in a conversation with anyone, or (2) Alice is in a conversation with another user Bob. In the first case, she sends a dummy message encrypted for herself to each chain that will come back to her own mailbox. We call these messages *loopback messages*. In the second case, Alice and Bob compute each other's choices of chains, and discover at which chain they will intersect. If there are multiple such chains, they break ties in a deterministic fashion. Then, Alice and Bob send the messages encrypted for the other person, which we call *conversation messages*, to their intersecting chain. They also send loopback messages on all other chains.

Security properties. We now argue the security informally. We present a more formal definition and arguments of privacy in Appendix B. Since both types of messages are encrypted for owners of mailboxes and the mix-net hides the origin of a message, the adversary cannot tell if a message going to Alice's mailbox is a loopback message or a conversation message sent by a different user. This means that the network pattern of all users is the same from the adversary's perspective: each user sends and receives exactly ℓ messages, each of which could be a loopback or a conversation message. As a result, the adversary cannot tell if a user is in a conversation or not. Moreover, we choose the chains such that every pair of users intersects at some chain (§5), meaning the probability that Alice is talking to a particular honest user is the same for all honest users. This hides the conversation metadata.

The analysis above, however, only holds if the adversary does not tamper with the messages. For instance, if the adversary drops Alice's message in a chain, then there are two possible observable outcomes in this chain: Alice receives (1) no message, meaning Alice is not in a conversation in this chain, or (2) one message, meaning someone intersecting with Alice at this chain is chatting with Alice. This information leakage breaks the security of XRD. We propose a new protocol called aggregate hybrid shuffle (§6) that efficiently defends against such an attack.

Scalability properties. Let n and N be the number of chains and servers in the network, respectively. Each user must send at least $\sqrt{n}$ messages to guarantee every pair of users intersect. To see why, fix ℓ, the number of chains a user selects. Those chains must connect a user Alice to all M users. Since the total number of messages sent by users is $M \cdot \ell$, each chain should handle $\frac{M \cdot \ell}{n}$ messages if we distribute the load evenly. We then need $\frac{M \cdot \ell}{n} \cdot \ell \geq M$ because the left hand side is the maximum number of users connected to the chains that Alice chose. Thus, $\ell \geq \sqrt{n}$. In §5, we present an approximation algorithm that uses $\ell \approx \sqrt{2n}$ to ensure all users intersect with each other while evenly distributing the work. This means that each chain handles $\approx \frac{\sqrt{2}M}{\sqrt{n}}$ messages, and thus XRD scales with the number of chains. If we set $n = N$ and each server appears in k chains for $k << \sqrt{N}$, which means $C(M,N) = \frac{k\sqrt{2}M}{\sqrt{N}} \to 0$ polynomially as $N \to \infty$ (§3.2). We show that k is logarithmic in N in §5.2.1.

5 XRD design

We now present the details of a XRD design that protects against an adversary that does *not* launch active attacks. We then describe modifications to this design that allows XRD to protect against active attacks in §6.

5.1 Mailboxes and mailbox servers

Every user in XRD has a mailbox that is publicly associated with her. In our design, we use the public key of a user as the identifier for the mailbox, though different public identifiers like e-mail addresses can work as well. The mailboxes are maintained by the mailbox servers, with simple put and get functionalities to add and fetch messages to a mailbox.

Algorithm 1 Mix server routing protocol

Server i is in chain x which contains k servers with its mixing key pair $(\mathsf{mpk}_i = g^{\mathsf{msk}_i}, \mathsf{msk}_i)$. In each round r, it receives a set of ciphertexts $\{c_i^j = (g^{x_j}, \mathsf{AEnc}(\mathsf{DH}(\mathsf{mpk}_i, x_j), r||\mathsf{x}, c_{i+1}^j))\}_{j \in [M]}$, either from an upstream server if $i \neq 1$, or from the users if $i = 1$.

1. **Decrypt and shuffle:** Compute $c_{i+1}^j = \mathsf{ADec}(\mathsf{DH}(g^{x_j}, \mathsf{msk}_i), r||\mathsf{x}, c_i^j)$ for each j, and randomly shuffle $\{c_{i+1}^j\}$.

2a. **Relay messages:** If $i < k$, then send the shuffled $\{c_{i+1}^j\}$ to server $i+1$.

2b. **Forward messages to mailbox:** If $i = k$, then each decrypted message is of the form $(\mathsf{pk}_u, \mathsf{AEnc}(\mathsf{s}, r||\mathsf{x}, m_u))$, where pk_u is the public key of a user u, s is a secret key, and m_u is a message for the user. Send the message to the mailbox server that manages mailbox pk_u.

5.2 Mix chains

XRD uses many parallel mix-nets to process the messages. We now describe their formation and operations.

5.2.1 Forming mix chains

We require the existence of an honest server in every chain to guarantee privacy. To ensure this property, we use public randomness sources [7, 50] that are unbiased and publicly available to randomly sample k servers to form a chain, similar to prior works [34, 52]. We set k large enough such that the probability that all servers are malicious is negligible. Concretely, the probability that a chain of length k consists only of malicious servers is f^k. Then, if we have n chains in total, the probability there exists a group of only malicious servers is less than $n \cdot f^k$ via a union bound. Finally, we can upper bound this to be negligible. For example, if we want this probability to be less than 2^{-64} for $f = 20\%$, then we need $k = 32$ for $n < 2000$. This makes k depend logarithmically on N. In XRD, we set $n = N$ for N servers, meaning each server appears in k chains on average.

We "stagger" the position of a server in the chains to ensure maximal server utilization. For instance, if a server is part of two chains, then it could be the first server in one chain and the second server in the other chain. This optimization has no impact on the security, as we only require the existence of an honest server in each group. This helps minimize the idle time of each server.

5.2.2 Processing user messages

After the chains form, each mix server i generates a *mixing key pair* $(\mathsf{mpk}_i = g^{\mathsf{msk}_i}, \mathsf{msk}_i)$, where msk_i is a random value in $\mathbb{Z}_p$. The public mixing keys $\{\mathsf{mpk}_i\}$ are made available to all participants in the network, along with the ordering of the keys in each chain. Now, each chain behaves as a mix-net [10]: users submit some messages onion-encrypted using the mixing keys (§5.3), and the servers decrypt and shuffle the messages in order. Algorithm 1 describes this protocol.

5.2.3 Server churn

Some servers may go offline in the middle of a round. Though XRD does not provide additional fault tolerance mechanisms, only the chains that contain failing servers are affected. Furthermore, the failing chains do not affect the security since they do not disturb the operations of other chains and the destination of the messages at the failing chain remains hidden to the adversary. Thus, conversations that use chains with no failing servers are unaffected. We analyze the empirical effects of server failures in §7.3.

5.3 Users

We now describe how users operate in XRD.

5.3.1 Selecting chains

XRD needs to ensure that all users' choices of chains intersect at least once, and that the choices are publicly computable. We present a scheme that achieves this property. Upon joining the network, every user is placed into one of $\ell+1$ groups such that each group contains roughly the same number of users, and such that the group of any user is publicly computable. This could be done, for example, by assigning each user to a pseudo-random group based on the hash of the user's public key. Every user in a group is connected to the same ℓ servers specified as follows. Let C_i be the ordered set of chains that users in group i are connected to. We start with $C_1 = \{1,\ldots,\ell\}$, and build the other sets inductively: For $i = 1,\ldots,\ell$, group $i+1$ is connected to $C_{i+1} = \{C_1[i], C_2[i], \ldots, C_i[i], C_i[\ell]+1, \ldots, C_i[\ell]+(\ell-i)\}$, where $C_x[y]$ is the y^{th} entry in C_x.

By construction, every group is connected to every other group: Group i is connected to group j via $C_i[j]$ for all $i < j$. As a result, every user in group i is connected to all others in the same group (they meet at all chains in C_i), and is connected to users in group j via chain $C_i[j]$.

To find the concrete value of ℓ, let us consider C_ℓ. The last chain of C_ℓ, which is the chain with the largest index, is $C_\ell[\ell] = \ell^2 - \sum_{i=1}^{\ell-1} i = \frac{\ell^2+\ell}{2}$. This value should be as close as possible to n, the number of chains, to maximize utilization. Thus, $\ell = \lceil\sqrt{2n+0.25}-0.5\rceil \approx \lceil\sqrt{2n}\rceil$. Given that $\ell \geq \sqrt{n}$ (§4), this is a $\sqrt{2}$-approximation.

5.3.2 Sending messages

After choosing the ℓ mix chains, the users send one message to each of the chosen chains as described in Algorithm 2. At a high level, if Alice is not talking with anyone, Alice generates ℓ loopback messages by encrypting dummy messages (e.g., messages with all zeroes) using a secret key known only to her, and submits them to the chosen chains. If she is talking with another user Bob, then she first finds where they intersect by computing the intersection of Bob's group and her group (§5.3.1). If there is more than one such chain, then she breaks the tie by selecting the chain with the smallest index. Alice then generates $\ell-1$ loopback messages and one encrypted message using a secret key that Alice and Bob shares. Finally, Alice sends the message for Bob to the intersecting chain, and sends the loopback messages to the other chains. Bob mirrors Alice's actions.

Algorithm 2 User conversation protocol

Consider two users Alice and Bob with key pairs $(\mathsf{pk}_A = g^{\mathsf{sk}_A}, \mathsf{sk}_A)$ and $(\mathsf{pk}_B = g^{\mathsf{sk}_B}, \mathsf{sk}_B)$ who are connected to sets of ℓ chains C_A and C_B (§5.3.1). The network consists of chains $1,\ldots,n$, each with k servers. Alice and Bob possess the set of mixing keys for each chain. Alice performs the following in round r.

1a. **Generate loopback messages:** If Alice is not in a conversation, then Alice generates ℓ loopback messages: $m_{\mathsf{x}} = (\mathsf{pk}_A, \mathsf{AEnc}(\mathsf{s}_A^{\mathsf{x}}, r||\mathsf{x}, 0))$ for $\mathsf{x} \in C_A$, where $\mathsf{s}_A^{\mathsf{x}}$ is a chain-specific symmetric key known only to Alice.

1b. **Generate conversation message:** If Alice is in a conversation with Bob, then she first computes the shared key $\mathsf{s}_{AB} = \mathsf{DH}(\mathsf{pk}_B, \mathsf{sk}_A)$, and the symmetric encryption key for Bob $\mathsf{s}_B = \mathsf{KDF}(\mathsf{s}_{AB}, \mathsf{pk}_B, r)$ where KDF is a secure key derivation function (e.g., HKDF [33]). Alice then generates the conversation message: $m_{\mathsf{x}_{AB}} = (\mathsf{pk}_B, \mathsf{AEnc}(\mathsf{s}_B, r||\mathsf{x}, \mathsf{msg}))$, where msg is the plaintext message for Bob and $\mathsf{x}_{AB} \in C_A \cap C_B$ is the first chain in the intersection. She also generates $\ell-1$ loopback messages m_{x} for $\mathsf{x} \in C_A, \mathsf{x} \neq \mathsf{x}_{AB}$.

2. **Onion-encrypt messages:** For each message m_{x}, let $c_{k+1} = m_{\mathsf{x}}$, and let $\{\mathsf{mpk}_i\}$ be the mixing keys for chain $\mathsf{x} \in C_A$. For $i = k$ to 1, generate a random value $x_i \in \mathbb{Z}_p$, and compute $c_i = (g^{x_i}, \mathsf{AEnc}(\mathsf{DH}(\mathsf{mpk}_i, x_i), r||\mathsf{x}, c_{i+1}))$. Send c_1 to chain x.

3. **Fetch messages:** At the end of the round, fetch and decrypt the messages in her mailbox, using ADec with matching $\mathsf{s}_A^{\mathsf{x}}$ or $\mathsf{s}_A = \mathsf{KDF}(\mathsf{s}_{AB}, \mathsf{pk}_A, r)$.

5.3.3 User churn

Like servers, users might go offline in the middle of a round, and XRD aims to provide privacy in such situations. However, the protocol presented thus far does not achieve this goal. If Alice and Bob are conversing and Alice goes offline without Bob knowing, then Alice's mailbox will receive Bob's message while Bob's mailbox will get one fewer message. Thus, by observing mailbox access counts and Alice's availability, the adversary can infer their communication.

To solve this issue, we require Alice to submit two sets of messages in round r: the messages for the current round r, and *cover messages* for round $r+1$. If Alice is not communicating with anyone, then the cover messages will be loopback messages. If Alice is communicating with another user, then one of the cover messages will be a conversation message indicating that Alice has gone offline.

If Alice goes offline in round τ, then the servers use the cover messages submitted in $\tau-1$ to carry out round τ. Now, there are two possibilities. If Alice is not in a conversation, then Alice's cover loopback messages are routed in round τ, and nothing needs to happen afterwards. If Alice is conversing with Bob, then at the end of round τ, Bob will get the message that Alice is offline via one of the cover messages. Starting from round $\tau+1$, Bob now sends loopback messages instead of conversation messages to hide the fact that Bob was talking with Alice in previous rounds. This could be used to end conversations as well. Malicious servers cannot fool Bob into thinking Alice has gone offline by replacing Alice's messages with her cover messages because the honest servers will ensure Alice's real messages are accounted for using our defenses described in §6.

6 Aggregate hybrid shuffle

Adversarial servers can tamper with the messages to leak privacy in XRD. For example, consider a mix-net chain where the first server is malicious. This malicious server can replace Alice's message with a message directed at Alice. Then, at the end of the mixing, the adversary will make one of two observations. If Alice was talking to another user Bob, Bob will receive one fewer message while Alice would receive two messages. The adversary would then learn that Alice was talking to Bob. If Alice is not talking to anyone on the tampered chain, then Alice would receive one message, revealing the lack of conversation on that chain.

Prior works [14, 55, 35, 52, 34] have used traditional verifiable shuffles [45, 25, 8, 27] to prevent these attacks. At a high level, verifiable shuffles allow the servers in the chain (one of which is honest) to verify the correctness of a shuffle of another server; namely, that the plaintexts underlying the outputs of a server is a valid permutation of the plaintexts underlying the inputs. Unfortunately, these techniques are computationally expensive, requiring many exponentiations.

In XRD, we make an observation that help us avoid traditional verifiable shuffles. For a meaningful tampering, the adversary necessarily has to tamper with the messages *before* they are shuffled by the honest server. Otherwise, the adversary does not learn the origins of messages. For example, after dropping a message in a server downstream from the honest server, the adversary might observe that Alice did not receive a message. The adversary cannot tell, however, whether the dropped message was sent by Alice or another user, and does not learn anything about Alice's communication pattern. (Intuitively, the adversarial downstream servers do not add any privacy in any case.) In this section, we describe a new form of verifiable shuffle we call *aggregate hybrid shuffle* (AHS) that allows us to take advantage of this fact. In particular, the protocol guarantees that the honest server will receive and shuffle all honest users' messages, or the honest server will detect that someone upstream (malicious servers or users) misbehaved. We will then describe how the honest server can efficiently identify all malicious participants who deviated from the protocol, without affecting the privacy of honest users. Table A in Appendix A summarizes the notations used in AHS.

6.1 Key generation with AHS

When the chain is created, the servers generate three key pairs: *blinding*, *mixing*, and *inner* key pairs. The inner keys are per-round keys, and each server i generates its own inner key pair $(\mathsf{ipk}_i = g^{\mathsf{isk}_i}, \mathsf{isk}_i)$. The other two keys are long-term keys, and are generated in order starting with the first server in the chain. Let $\mathsf{bpk}_0 = g$. Starting with server 1, server $i = 1,\ldots,k$ generates $(\mathsf{bpk}_i = \mathsf{bpk}_{i-1}^{\mathsf{bsk}_i}, \mathsf{bsk}_i)$ and $(\mathsf{mpk}_i = \mathsf{bpk}_{i-1}^{\mathsf{msk}_i}, \mathsf{msk}_i)$ in order. In other words, the base of the public keys of the server i is $\mathsf{bpk}_{i-1} = g^{\prod_{a<i}\mathsf{bsk}_a}$. The public mixing key of the last server, for example, would be $\mathsf{mpk}_k = \mathsf{bpk}_{k-1}^{\mathsf{msk}_k} = g^{\mathsf{msk}_k \cdot \prod_{a<k}\mathsf{bsk}_a}$. Each server also has to prove to all other servers that it knows the private keys that match the public keys in zero-knowledge [9]. All public keys are made available to all servers and users.

6.2 Sending messages with AHS

Once the servers generate the keys, user Alice can submit a message to a chain. To do so, Alice now employs a double-enveloping technique to encrypt her message [26]: she first onion-encrypts her message for all servers using the inner keys, and then onion-encrypts the result with the mixing keys. Let *inner ciphertext* be the result of the first onion-encryption, and *outer ciphertext* be the final ciphertext. The inner ciphertexts are encrypted using $\prod_i \mathsf{ipk}_i$ as the public key, which allows users to onion-encrypt in "one-shot": i.e., $e = (g^y, \mathsf{AEnc}(\mathsf{DH}(\prod_i \mathsf{ipk}_i, y), r, m))$ in round r with message m and a random y. Without y, one must know all $\{\mathsf{isk}_i\}$ to compute $\mathsf{DH}(\prod_i \mathsf{ipk}_i, y)$, which makes this a "one-shot" onion-encryption. To generate the outer ciphertext for chain x, Alice performs the following.

1. Generate her outer Diffie-Hellman key: a random $x \in \mathbb{Z}_p$ and (g^x, x).
2. Generate a NIZK that proves she knows x that matches g^x (using knowledge of discrete log proof [9]).
3. Let $c_{k+1} = e$, and let $\{\mathsf{mpk}_i\}$ for $i \in [k]$ be the mixing keys of the servers in the chain. For $i = k$ to 1, compute $c_i = \mathsf{AEnc}(\mathsf{DH}(\mathsf{mpk}_i, x), r||\mathsf{x}, c_{i+1})$.

$c = (g^x, c_1)$ is the final outer ciphertext. This is nearly identical to Algorithm 2, except that the user does not generate a fresh pair of Diffie-Hellman keys for each layer of encryption. To submit the message, Alice sends c and the NIZK to all servers in the chain.

6.3 Mixing with AHS

Before mixing begins in round r, the servers in chain x have $c^j = (X_1^j = g^{x_j}, c_1 = \mathsf{AEnc}(\mathsf{DH}(\mathsf{mpk}_1, x_j), r||\mathsf{x}, c_2^j))$ for user j. The servers first verify all NIZKs the users submit, and agree on the inputs for this round. This can be done, for example, by sorting the users' ciphertexts, hashing them using a cryptographic hash function, and then comparing the hashes. Then, starting with server 1, server $i = 1,\ldots,k$ perform the following:

1. **Decrypt and shuffle:** Similar to Algorithm 1, decrypt each message. Each message is of the form $(X_i^j, c_i^j = \mathsf{AEnc}(\mathsf{DH}(\mathsf{mpk}_i, x_j), r||\mathsf{x}, c_{i+1}^j)$. Thus, $(b^j, c_{i+1}^j) = \mathsf{ADec}(\mathsf{DH}(X_i^j, \mathsf{msk}_i), r||\mathsf{x}, c_i^j)$. If any decryption fails (i.e., $b^j = 0$ for some j), then mixing halts and the server can start the blame protocol described in §6.4. Randomly shuffle $\{c_{i+1}^j\}$.
2. **Blind and shuffle:** Blind the users' Diffie-Hellman keys $\{X_i^j\}$ using the blinding key: $X_{i+1}^j = (X_i^j)^{\mathsf{bsk}_i}$ for each j. Then, shuffle the keys using the same permutation as the one used to shuffle the ciphertexts.
3. **Generate zero-knowledge proof:** Generate a proof that $(\prod_j X_i^j)^{\mathsf{bsk}_i} = \prod_j X_{i+1}^j$ by generating a NIZK that shows $\log_{\prod_j (X_i^j)}(\prod_j X_{i+1}^j) = \log_{\mathsf{bpk}_{i-1}} \mathsf{bpk}_i (= \mathsf{bsk}_i)$. Send the NIZK with the shuffled $\{X_{i+1}^j\}$ to all other servers in the chain. All other servers verify this proof using $\{X_i^j\}$ they received previously, $\{X_{i+1}^j\}$, bpk_{i-1}, and bpk_i.
4. **Forward messages:** If $i < k$, then send the shuffled $\{(X_{i+1}^j, c_{i+1}^j)\}$ to server $i+1$.

When the last server finishes shuffling and no server reports any errors during mixing, our protocol guarantees that the honest server mixed all the honest users' messages successfully. At this point, the servers reveal their private per-round inner keys $\{\mathsf{isk}_i\}$. With this, the last server can decrypt the inner ciphertexts to recover the users' messages.

Analysis. We first argue correctness of AHS (i.e., every message is successfully delivered if every participant followed the protocol) by showing that the encryption and decryption keys match at all layers. Consider the key user j used to encrypt the message for server i and the Diffie-Hellman key server i receives. User j encrypts the message using the key $\mathsf{DH}(\mathsf{mpk}_i, x_j) = g^{x_j \cdot \mathsf{msk}_i \prod_{a<i} \mathsf{bsk}_a}$. The Diffie-Hellman key server i receives is $X_i^j = g^{x_j \cdot \prod_{a<i} \mathsf{bsk}_a}$. The key exchange then results in $\mathsf{DH}(X_i^j, \mathsf{msk}_i) = g^{\mathsf{msk}_i \cdot x_j \prod_{a<i} \mathsf{bsk}_a}$, which is the same as the one the user used.

The scheme protects against honest-but-curious adversaries (i.e., does not reveal anything about the permutation used to shuffle the messages), as the inputs and outputs of a server look random: If decisional Diffie-Hellman is hard, then $g^{x \cdot \mathsf{bsk}_i}$ is indistinguishable from a random value given g^x and g^{bsk_i} for random x and bsk_i. Thus, by observing $\{g^{x_j}\}$ (input) and $\{g^{x_{\pi(j)} \cdot \mathsf{bsk}_i}\}$ (output) of an honest server where π is a random permutation, the adversary cannot learn anything about the relationships between the inputs and outputs.

We now provide a high level analysis that the honest server will always detect upstream servers tampering with honest users' messages. The detailed proof is in Appendix A. Let server h be the honest server. First, since we only need to consider upstream adversaries, we will simplify the problem, and view all upstream malicious servers as one collective server with private blinding key $\mathsf{bsk}_A = \sum_{i<h} \mathsf{bsk}_i$. For the adversary to successfully tamper, it must generate $\{X_h^j\}$ such that $(\prod X_1^j)^{\mathsf{bsk}_A} = \prod X_h^j$; otherwise it would fail the NIZK verification in step 3 of §6.3. Let $X_T \neq \emptyset$ be the set of honest users whose messages were tampered. The adversary needs to know the keys used for authenticated decryption to generate valid ciphertexts that differ from the users' ciphertexts. However, the adversary cannot compute $((g^{x_j})^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$ for $j \in X_T$ (the keys used for authenticated encryption), since x_j and msk_h are unknown random values and Diffie-Hellman is hard. Thus, to tamper with messages undetected, the adversary needs to change the users' Diffie-Hellman keys (i.e., $X_h^j \neq (X_1^j)^{\mathsf{bsk}_A}$ for $j \in X_T$), such that it can compute the keys used for authenticated decryption (i.e., $(X_h^j)^{\mathsf{msk}_h}$ for $j \in X_T$).

In the beginning of a round, the adversary controlled users have to prove their knowledge of discrete logs of their Diffie-Hellman keys after seeing the honest users' keys. The adversarial user are thus forced to generate keys independently of the honest users' input. Then, the adversary's goal is essentially to find $\{X_h^j\}_{j \in X_T}$ such that $(\prod_{j \in X_T} X_1^j)^{\mathsf{bsk}_A} = \prod_{j \in X_T} X_h^j$, with $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$. Assume the adversary is successful. Then, it could compute $((\prod_{j \in X_T} X_1^j)^{\mathsf{bsk}_A})^{\mathsf{msk}_h} = \prod_{j \in X_T} (X_h^j)^{\mathsf{msk}_h}$, since it knows $(X_h^j)^{\mathsf{msk}_h}$ for $j \in X_T$ (recall that these are the keys used for authenticated decryption, which the adversary must know). This means that the adversary computed $((\prod_{j \in X_T} X_1^j)^{\mathsf{bsk}_A})^{\mathsf{msk}_h} = g^{\mathsf{msk}_h \cdot \mathsf{bsk}_A \cdot \sum_{j \in X_T} x_j}$ only given $\{g^{x_j}\}_{j \in X_T}$, bsk_A, and $(g^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$, where $\{x_j\}_{j \in X_T}$ and msk_h are random values independent of bsk_A. This breaks the Diffie-Hellman assumption, and thus the adversary must not be able to tamper with messages undetected.

6.4 Blame protocol

There are two ways an honest server can detect misbehavior: a NIZK fails to verify or an authenticated decryption fails. If a malicious user cannot generate a correct NIZK in step 2 in §6.2 or if a malicious server misbehaves and cannot generate a correct NIZK in step 3 in §6.3, then the misbehavior is detected and the adversary is immediately identified. In the case where a server finds some misauthenticated ciphertexts, the server can start a blame protocol that allows the server to identify who misbehaved. The protocol guarantees that users are identified if and only if they purposefully sent misauthenticated ciphertexts. In addition, the protocol ensures that honest users remain private in all cases, even if malicious servers try to falsely accuse honest users.

Once server h identifies an misauthenticated ciphertext, it starts the blame protocol by revealing the problem ciphertext (X_h^j, c_h^j). Then, the servers execute the following:

1. For $i = h-1, \ldots, 1$, the servers reveal X_i^j that matches X_{i+1}^j (i.e., $(X_i^j)^{\mathsf{bsk}_i} = X_{i+1}^j$). Each server proves to all other servers it calculated X_{i+1}^j correctly by showing that $\log_{X_i^j}(X_{i+1}^j) = \log_{\mathsf{bpk}_{i-1}}(\mathsf{bpk}_i)(= \mathsf{bsk}_i)$ with a NIZK [9].

2. For $i = h-1, \ldots, 1$, the servers reveal c_i^j that matches c_{i+1}^j (i.e., $c_i^j = \mathsf{AEnc}(\mathsf{DH}(X_i^j, \mathsf{msk}_i), r||\mathsf{x}, c_{i+1}^j)$. Each server proves it correctly decrypted the ciphertext by revealing the key used for decryption $k_i^j = (X_i^j)^{\mathsf{msk}_i}$, and showing that $\log_{X_i^j}(k_i^j) = \log_{\mathsf{bpk}_{i-1}}(\mathsf{mpk}_i)$ with a NIZK. The other servers can verify the correctness of the decryption operation by checking the NIZKs and decrypting the ciphertext themselves.

3. All servers check that c_1^j revealed by the first server matches the user submitted ciphertext (§6.2).

4. Similar to step 2, server h (the accusing server) reveals its Diffie-Hellman exchanged key $k_h = (X_h^j)^{\mathsf{msk}_h}$, and shows that $\log_{X_h^j}(k_h) = \log_{\mathsf{bpk}_{h-1}}(\mathsf{mpk}_h)$. All servers verify that $\mathsf{ADec}(k_h, r||\mathsf{x}, c_h^j)$ fails.

If there are multiple problem ciphertexts, the blame protocol can be carried out in parallel for each ciphertext. Steps 1 and 2 can be done simultaneously as well. If the servers successfully carry out the blame protocol, then they have identified actively malicious users. At this point, those ciphertexts are removed from the set, and the upstream servers are required to repeat the AHS protocol; since the accusing servers have already removed all bad ciphertexts, the servers just have to repeat step 3 of §6.3 to show the keys were correctly computed. If any of the above steps fail, then the servers delete their private inner keys.

Analysis. The accusing server and the upstream servers are required to reveal the exchanged key used to decrypt the ciphertexts, and the correctness of the key exchange is proven through the two NIZKs in step 1 and step 2. All servers can use the revealed keys to ensure that the submitted original ciphertext decrypts to the problem ciphertext. Since the outer ciphertext behaves as a commitment to all layers of encryption (§3), the servers get a verifiable chain of decryption starting with the outer ciphertext to the problem ciphertext if a user submits misauthenticated ciphertext. Moreover, if an honest user submits a correctly authenticated ciphertext, she will never be accused successfully, since an honest user's ciphertext will authenticate at all layers. Thus, a user is identified if and only if she is malicious.

Importantly, the users' privacy is protected even after a false accusation. After a malicious server accuses an honest user, the malicious server learns either the outer ciphertext (if the server is upstream of server h) or the inner ciphertext (if the server is downstream of server h) the user sent. In either case, the message remains encrypted for the honest server, by the mixing key in the former case or by the inner key in the latter case. The blame protocol will fail when the malicious server fails to prove that the honest user's ciphertext is misauthenticated, and the ciphertext will never be decrypted. As such, the adversary never learns the final destination of the user's message, and XRD protects the honest users' privacy.

7 Implementation and evaluation

To evaluate XRD, we wrote a prototype in approximately 4,000 lines of Go. We used the NIST P-256 elliptic curve [2] for our cryptographic group, and used AES with SHA-256-based key derivation function [33] and HMAC [32] for our authenticated encryption scheme. The servers communicated using streaming gRPC over TLS. Our prototype assumes that the servers' and users' public keys, and the public randomness for creating the chains are provided by a higher level functionality. Finally, we set the number of chains n equal to the number of servers N (§5.2.1).

In this section, we investigate the cost of users and the performance of XRD for different network configurations. For majority of our experiments, we assumed $f = 0.2$ (i.e., 80% of the servers are honest) unless indicated otherwise. We used 256 byte messages, similar to evaluations of prior systems [53, 52, 5]; this is about the size of a standard SMS message or a Tweet. We used c4.8xlarge instances on Amazon EC2 for our experiments, which has 36 Intel Xeon E5-2666 CPUs with 60 GB of memory and 10 Gbps links.

We compare the results against four prior systems: Stadium [52], Karaoke [37], Atom [34], and Pung [5, 4]. For Stadium and Karaoke, we report the performance for $e^{\varepsilon} = 10$ and $e^{\varepsilon} = 4$, respectively, (meaning the probability of Alice talking with Bob is within $10\times$ and $4\times$ the probability of Alice talking with any other user) and allow up to 10^8 rounds of communication with strong security guarantees ($\delta < 10^{-4}$), which are the parameters the used for evaluation in their papers. We show results for Pung with XPIR (used in the original paper [5]) and with SealPIR (used in the follow-up work [4]) when measuring user overheads, and only with XPIR when measuring end-to-end latency. We do this because SealPIR significantly improves the client performance and enables more efficient batch messaging, but introduces extra server overheads in the case of one-to-one messaging.

As mentioned in §2, the four systems scale horizontally, but offer different security properties. To summarize, Stadium and Karaoke provide differential privacy guarantees against the same adversary assumed by XRD. Thus, their users can send a limited number of sensitive messages with strong privacy, while XRD users can do so for unlimited messages. Atom provides cryptographic sender anonymity [46] under the same threat model. Finally, Pung provides messaging with cryptographic privacy against an adversary who can compromise all servers rather than a fraction of servers. To the best of our knowledge, we are not aware of any scalable private messaging systems that offer the same security guarantee under a similar threat model to XRD.

7.1 User costs

We first characterize computation and bandwidth overheads of XRD users using a single core of a c4.8xlarge instance. In order to ensure that every pair of users intersects, each user sends $\sqrt{2N}$ messages (§5.3.1). This means that

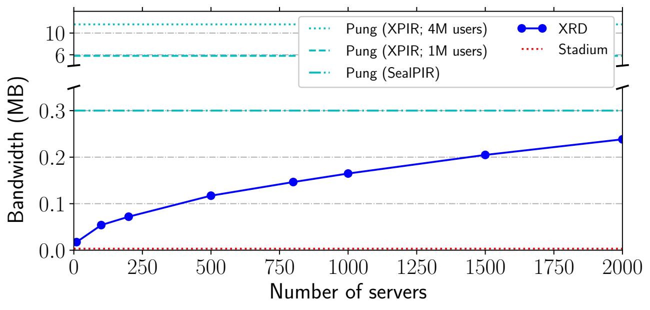


Figure 2: Required user bandwidth per round as a function of number of servers in the network.

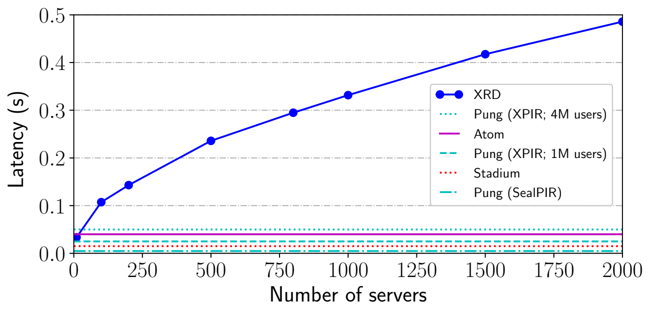


Figure 3: Required user computation as a function of number of servers with a single core. The computation could easily be parallelized with more cores for XRD.

the overheads for users increase as we add more servers to the network, as shown in Figure 2 and 3. This is a primary weakness of XRD, since our horizontal scalability comes at a higher cost for users. Still, the cost remains reasonable even for large numbers of servers. With 2,000 servers, each user must submit about 238 KB of data. For 1 minute rounds, this translates to about 40 Kbps of bandwidth requirement. A similar trend exists for computation overhead as well, though it remains relatively small: it takes less than 0.5s with fewer than 2,000 servers in the network. Computation could also be easily parallelized with more cores, since users can generate the messages for different chains independently. The cover messages make up half of the client overhead (§5.3.3).

User costs in prior works do not increase with the number of servers. Still, Pung with XPIR incurs heavy user bandwidth overheads due to the cost of PIR. With 1 million users, Pung users transmit about 5.8 MB, which is about 25× worse than XRD when there are fewer than 2,000 servers. Moreover, per user cost of XPIR is proportional to the total number of users: the bandwidth cost increases to 11 MB of bandwidth for 4 million users. The SealPIR variant, however, is comparable to that of XRD, as the users can compress the communication using cryptographic techniques. Stadium, Karaoke, and Atom incur minimal user bandwidth cost, with less than a kilobyte of bandwidth overhead (only Stadium is shown Figure 2). Thus, for users with heavily limited resources, prior works can be more desirable than XRD.

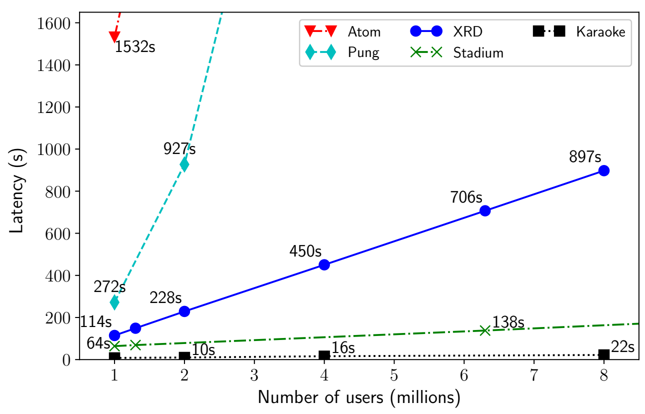


Figure 4: End-to-end latency of XRD and prior systems with varying numbers of users with 100 servers.

7.2 End-to-end latency

Experiment setup. To evaluate the end-to-end performance, we created a testbed consisting of up to 200 c4.8xlarge instances. We ran the instances within the same data center to avoid bandwidth costs, but added 40-100ms of round trip latency between servers using the Linux `tc` command to simulate a more realistic distributed network. Our evaluations consider all parts of the AHS protocol (§6), but assume all users are following the protocol. We then evaluate the blame protocol (§6.4) separately.

We used many c4.8xlarge instances to simulate millions of users, and also used ten more c4.8xlarge instances to simulate the mailboxes. We generate all users' messages before the round starts, and measure the critical path of our system by measuring the time between the last user submitting her message and the last user downloading her message.

We estimate the latency of Pung with M users and N servers by evaluating it on a single c4.8xlarge instance with M/N users. This is the best possible latency Pung can achieve because (1) Pung is embarrassingly parallel, so evenly dividing users across all the servers should be ideal [5, §7.3], and (2) we are ignoring the extra work needed for coordination between the servers (e.g., for message replication). For Stadium, we report the latency when the length of each mix chain is nine servers. For Karaoke, we report the numbers reported in their paper.

We focus on the following questions in this section, and compare against prior work:

- What is the end-to-end latency of XRD, and how does it change with the number of users?
- How does XRD scale with more servers?
- What is the effect of f, the fraction of malicious servers, on latency?
- How fast is the blame protocol?

Number of users. Figure 4 shows the end-to-end latency of XRD and prior works with 100 servers. XRD was able to handle 2 million users in 228s, and the latency scales linearly

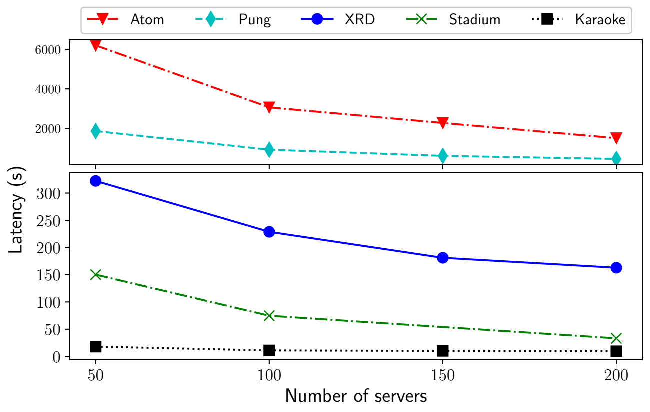


Figure 5: End-to-end latency of XRD for varying numbers of servers with 2 million users. We show Pung and Atom on a different time scale.

with the number of users. This is 13.3× and 4× faster than Atom and Pung, and 3× and 22.8× worse than Stadium and Karaoke for the same deployment scenario. Though processing a single message in XRD is faster than doing so in Stadium (since Stadium relies on verifiable shuffle, while XRD uses AHS), the overall system is still slower. This is because each XRD user submits many messages. For example, each user submits 15 messages with 100 servers, which is almost equivalent to adding 15 users who each submit one message. Unfortunately, the performance gap would grow with more servers due to each user submitting more messages (the rate at which the gap grows would be proportional to $\sqrt{2N}$). While XRD cannot provide the same performance as Stadium or Karaoke with large numbers of users and servers, XRD can provide stronger cryptographic privacy.

When compared to Pung, the speed-up increases further with the number of users since the latency of Pung grows superlinearly. This is because the server computation per user increases with the number of users. With 4 million users, for example, XRD is 8× faster. For Atom, the latency increases linearly, but with higher slope. This is due to its heavy reliance on expensive public key cryptography and long routes for the messages (over 300 servers).

Scalability. Figure 5 shows how the latency decreases with the number of servers with 2 million users.We experimented with up to 200 servers, and observed the expected scaling pattern: the latency of XRD reduces as $\sqrt{2/N}$ with N servers (§4). In contrast, prior works scale as $1/N$, and thus will outperform XRD with enough servers. Still, because XRD employs more efficient cryptography, XRD outperforms Atom and Pung with less than 200 servers.

To estimate the performance of larger deployments, we extrapolated our results to more servers. We estimate that XRD can support 2 million users with 1,000 servers in about 84s, while Stadium and Karaoke can do so in about 8s and 2s, respectively. (At this point, the latency between servers would be the dominating factor for Stadium and Karaoke.) This

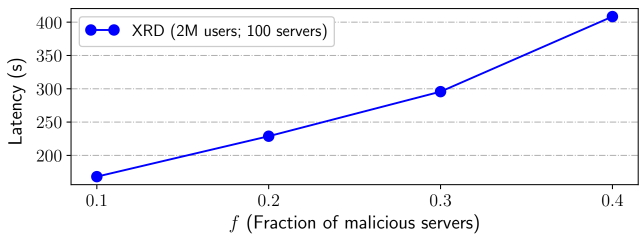


Figure 6: Latency of XRD for different values of f.

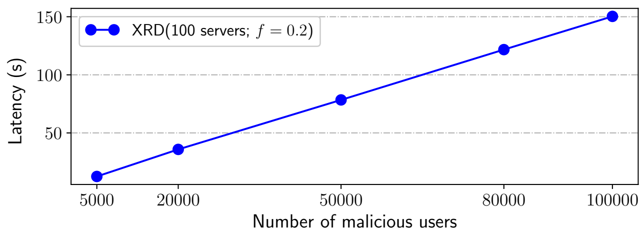


Figure 7: Latency of blame protocol.

gap increases with more users, as described previously. For Atom and Pung, we estimate that the latency would be comparable to XRD with about 3,000 servers and 1,000 servers in the network, respectively, for 2 million users. Pung would need more servers with more users to catch up to XRD due to the superlinear increase in latency with the number of users.

Impact of f. During setup, the system administrator should make a conservative estimate of f to form the chains. Larger f affects latency because it increases in the length of the chains k (§5.2.1). Concretely, with $n = 100$, k must satisfy $100 \cdot f^k < 2^{-64}$. Thus, $k > \frac{\log(2^{-64}/100)}{\log(f)}$, which means that the length of a chain (and the latency) grows as a function of $\frac{-1}{\log(f)}$. Figure 6 demonstrates this effect. The latency grows slowly for $f < 0.5$. This function, however, grows rapidly when $f >> 0.5$, and thus the latency would be significantly worse when considering larger values of f.

Atom would experience the same effect since its mix chains are created using the same strategy as XRD. Karaoke also experiences similar increase in latency with f, as every message must be routed through a number of servers proportional to $|\frac{1}{\log f}|$ as well. Stadium would face more significant increase in latency with f as its mix chains similarly get longer with f, and the length of the chains has a superlinear effect on the latency due to zero-knowledge proof verification [52, §10.3]. The latency of Pung does not increase with f since it already assumes $f = 1$.

Blame protocol. Malicious users could send misauthenticated ciphertexts to trigger the blame protocol, and slow down the system. Since malicious users' messages are removed as soon as servers find them, the users cause the most slowdown when the misauthenticated ciphertexts are discovered at the last server. The performance of the blame protocol also depends on the number of malicious users. In Figure 7, we therefore show the latency of the blame protocol

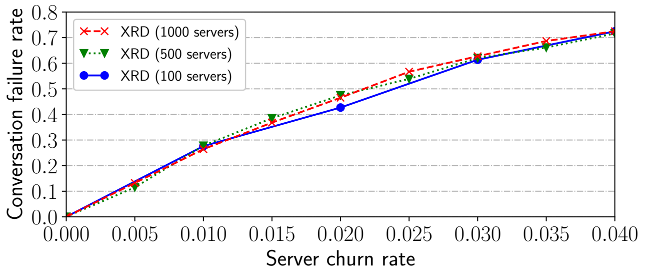


Figure 8: Fraction of conversations that fail in a given round due to server failures for different server churn rates.

as a function of the number of malicious users in a chain of 32 servers when the last server detects misbehavior. The blame protocol requires two discrete log equivalence proofs and decryption per user for each layer of encryption (§6.4).

Concretely, if 5,000 users misbehave in a chain, the blame protocol takes about 13s. This cost increases linearly with the number of users: if 100,000 users misbehave in a chain (which corresponds to approximately a third of all users being malicious with 100 servers and 2 million users in the network), the protocol takes about 150s. As a result, the overall round would take 378s, instead of 228s, for 2 million users. Still, this is 8× and 2.4× faster than Atom and Pung. In practice, the blame protocol could be faster since the honest server is likely to be not the last server. For example, if the honest server is the 16th server, then the blame protocol would take half the time. The overall round would then take around 303s.

While this is a significant increase in latency, malicious users are removed from the network once servers identify them. To cause serious slowdowns across many rounds, the adversary needs to constantly create new malicious users. Then, by employing defenses against Sybil attacks (e.g., imposing a small cost for user registration such as proof-of-work [23] or CAPTCHA [54]), we could limit the effectiveness of an adversary.

Malicious users can have varying degree of impact in prior systems. Pung and Karaoke are not affected by malicious users, and Stadium has a blame protocol similar to XRD to handle malicious users. Atom, however, is the significantly affected: a single malicious user can launch a DoS attack in the faster variant used for comparison in this paper [34, §4.4]. The slower variant of Atom that can handle adversarial users is at least 4× slower [34, §6], meaning it would be at least 30× slower than XRD overall.

7.3 Availability

To estimate the effect of server churn on a XRD network, we simulated deployment scenarios with 2 million users and different numbers of servers. We assumed that all users were in a conversation, and show the fraction of the users whose conversation messages did not reach their partner in Figure 8. For example, if 1% of the servers fail in a given round (comparable to server churn rate in Tor [1]), then we expect about 27% of the conversations to experience failure, and the end-points would have to resend their conversation messages. Unfortunately, the failure rate quickly increases with the server churn rates, reaching 70% with 4% server failures, as more chains contain at least one failing server. Thus, it would be easy for the adversary who controls a non-trivial fraction of the servers to launch a denial-of-service attack. Addressing this concern remains important future work.

When compared to Pung, the availability guarantees can be significantly worse, assuming Pung replicates all users' messages across all servers. In this case, the conversation failure rate would be equal to the server churn rate with users evenly distributed across all Pung servers, and the users connected to the failing servers could be rerouted to other servers to continue communication. Atom can tolerate any fraction γ of the servers failing using threshold cryptography [17], but the latency grows with γ. For example, to tolerate $\gamma = 1\%$ servers failing, we estimate that Atom would be about 10% slower [34, Appendix B]. Stadium and Karaoke, however, are more affected by server churn. Stadium uses two layers of parallel mix-nets, and fully connects the chains across the two layers. As a result, even one server failure would cause the whole system to come to a halt. (Stadium does not provide a fault recovery mechanism, and the security implications of continuing the protocol without the failing chains are not analyzed [52].) Similarly, Karaoke uses layers of interconnected mixing servers, and a single server failure results in the failure of the whole network.

8 Conclusion

XRD provides a unique design point in the space of metadata private communication systems by achieving cryptographic privacy and horizontal scalability using efficient cryptographic primitives. XRD organizes the servers into multiple small chains that process messages in parallel, and can scale easily with the number of servers by adding more chains. We hide users' communication patterns by ensuring every user is equally likely to be talking to any other user, and hiding the origins of users' messages through mix-nets. We then efficiently protect against active attacks using a novel technique called aggregate hybrid shuffle. Our evaluation on a network of 100 servers demonstrates that XRD can support 2 million users in 228 seconds, which is 13× and 4× faster than Atom and Pung, two prior systems with cryptographic privacy guarantees.

Acknowledgements

We thank Jun Wan, Daniel Grier, Ling Ren, Alin Tomescu, Zack Newman and Kyle Hogan for valuable feedback and discussion during this project. We also thank Paul Grubbs for pointing out a flaw in our initial proof. Finally, we thank the anonymous reviewers of NSDI, and our shepherd Raluca Ada Popa. This work was partially supported by a National Science Foundation grant (1813087).

References

[1] Tor metrics portal. https://metrics.torproject.org.

[2] Mehmet Adalier. Efficient and secure elliptic curve cryptography implementation of curve p-256. 2015.

[3] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *PETS*, 2016(2):155–174, 2016.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 1011–1028.

[5] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, pages 551–569, GA, 2016. USENIX Association.

[6] Mihir Bellare, Phillip Rogaway, and David Wagner. Eax: A conventional authenticated-encryption mode. Cryptology ePrint Archive, Report 2003/069, 2003. `https://eprint.iacr.org/2003/069`.

[7] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On Bitcoin as a public randomness source. `https://eprint.iacr.org/2015/1015.pdf`, 2015.

[8] Justin Brickell and Vitaly Shmatikov. Efficient anonymity-preserving data collection. In *KDD*, pages 76–85, New York, NY, USA, 2006. ACM.

[9] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. Technical report, 1997.

[10] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, February 1981.

[11] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, March 1988.

[12] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, November 1998.

[13] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE Symposium on Security and Privacy*, pages 321–338, May 2015.

[14] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable anonymous group messaging. In *CCS*, pages 340–350, New York, NY, USA, 2010. ACM.

[15] George Danezis, Roger Dingledine, David Hopwood, and Nick Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *IEEE Symposium on Security and Privacy*, pages 2–15, 2003.

[16] George Danezis and Andrei Serjantov. Statistical disclosure or intersection attacks on anonymity systems. In Jessica Fridrich, editor, *Information Hiding*, pages 293–308, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[17] Alfredo De Santis, Yvo Desmedt, Yair Frankel, and Moti Yung. How to share a function securely. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 522–533, New York, NY, USA, 1994. ACM.

[18] Tim Dierks and Eric Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, RFC Editor, August 2008. `http://www.rfc-editor.org/rfc/rfc5246.txt`.

[19] Roger Dingledine. One cell is enough to break tor's anonymity. `https://blog.torproject.org/one-cell-enough-break-tors-anonymity`, February 2009.

[20] Roger Dingledine and Nick Mathewson. Anonymity loves company: Usability and the network effect.

[21] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, pages 303–320. USENIX Association, August 2004.

[22] Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'06, pages 1–12, Berlin, Heidelberg, 2006. Springer-Verlag.

[23] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *CRYPTO*, pages 139–147, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[24] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *CCS*, CCS '02, pages 193–206, New York, NY, USA, 2002. ACM.

[25] Jun Furukawa and Kazue Sako. An efficient scheme for proving a shuffle. In *CRYPTO*, pages 368–387. Springer-Verlag, 2001.

[26] Philippe Golle, Sheng Zhong, Dan Boneh, Markus Jakobsson, and Ari Juels. Optimistic mixing for exit-polls. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 451–465. Springer, 2002.

[27] Jens Groth and Steve Lu. Verifiable shuffle of large size ciphertexts. In *PKC*, pages 377–392, 2007.

[28] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO*, pages 66–97. Springer International Publishing, 2017.

[29] Amir Houmansadr and Nikita Borisov. The need for flow fingerprints to link correlated network flows. In Emiliano De Cristofaro and Matthew Wright, editors, *Privacy Enhancing Technologies*, pages 205–224, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[30] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. Users get routed: Traffic correlation on Tor by realistic adversaries. In *CCS*, November 2013.

[31] Dogan Kedogan, Dakshi Agrawal, and Stefan Penz. Limits of anonymity in open environments. In Fabien A. P. Petitcolas, editor, *Information Hiding*, pages 53–69, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[32] H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication. RFC 2104, RFC Editor, February 1997.

[33] H. Krawczyk and P. Eronen. Hmac-based extract-and-expand key derivation function (hkdf). RFC 5869, RFC Editor, May 2010.

[34] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 406–422, New York, NY, USA, 2017. ACM.

[35] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. In *PETS*, volume 2016, pages 115–134, 2015.

[36] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962, RFC Editor, June 2013.

[37] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Fast and strong metadata privacy with low noise. In *OSDI*, Carlsbad, CA, 2018. USENIX Association.

[38] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *OSDI*, 2016.

[39] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. In *SIGCOMM*, pages 303–314, New York, NY, USA, 2013. ACM.

[40] Moxie Marlinspike and Trevor Perrin. Signal specifications. `https://https://signal.org/docs/`.

[41] Susan E. McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. Investigating the computer security practices and needs of journalists. In *USENIX Security*, pages 399–414, Washington, D.C., August 2015. USENIX Association.

[42] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, Washington, D.C., 2015. USENIX Association.

[43] Steve Mills. Defining the delicate and often difficult relationship between reporters and sources. `https://www.propublica.org/article/ask-propublica-illinois-reporters\-and-sources-relationship`, April 2018.

[44] Prateek Mittal and Nikita Borisov. Shadowwalker: Peer-to-peer anonymous communication using redundant structured topologies. In *CCS*, CCS '09, pages 161–172, New York, NY, USA, 2009. ACM.

[45] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *CCS*, pages 116–125, New York, NY, USA, 2001. ACM.

[46] Andreas Pfitzmann and Marit Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. August 2010.

[47] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *USENIX Security Symposium*, pages 1199–1216, Vancouver, BC, 2017. USENIX Association.

[48] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli Maurer, editor, *EUROCRYPT*, pages 387–398, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[49] Michael K. Reiter and Aviel D. Rubin. Anonymous web transactions with crowds. *Communications of the ACM*, 42(2):32–48, February 1999.

[50] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *IEEE Symposium on Security and Privacy*, pages 444–460, May 2017.

[51] Alin Tomescu and Srinivas Devadas. Catena: Efficient non-equivocation via bitcoin. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 393–409, May 2017.

[52] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *SOSP*, SOSP '17, pages 423–440, New York, NY, USA, 2017. ACM.

[53] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *SOSP*, pages 137–152. ACM, 2015.

[54] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. Captcha: Using hard ai problems for security. In Eli Biham, editor, *EUROCRYPT*, pages 294–311, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[55] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *OSDI*, pages 179–182, Hollywood, CA, 2012. USENIX Association.

A Security of aggregate hybrid shuffle

The adversary's goal is to have an upstream server successfully tamper with some messages without getting detected by the honest server. To model this, we consider the following security game between three parties: the client, the adversary, and the verifier. All parties are given the total number of users M. The client controls users in set $X_H \subset [M]$ (this models the honest users), and the adversary controls users in set $X_A = [M] \setminus X_H$. In addition, the adversary controls servers $1, \ldots, h-1$, and the verifier controls server h (i.e., the honest server). To simplify the presentation, we assume the adversary uses the identity permutation for all servers, but it is easy to adapt this proof to any arbitrary permutation. Table A lists the notation used in aggregate hybrid shuffle, and summarizes their purposes.

1. The adversary sends the client and the verifier the public keys bpk_i and mpk_i and ipk_i for $i = 1, \ldots, h-1$. It also generates NIZKs to prove that it knows the values $\mathsf{bsk}_i = \log_{\mathsf{bpk}_{i-1}}(\mathsf{bpk}_i)$ and $\mathsf{msk}_i = \log_{\mathsf{bpk}_{i-1}}(\mathsf{mpk}_i)$ for $i = 1, \ldots, h-1$, where $\mathsf{bpk}_0 = g$. It sends the public keys and NIZKs to the verifier.
2. The verifier verifies the NIZKs. The verifier then generates the key pairs $(\mathsf{bpk}_h = \mathsf{bpk}_{i-1}^{\mathsf{bsk}_h}, \mathsf{bsk}_h)$, $(\mathsf{mpk}_h = \mathsf{bpk}_{i-1}^{\mathsf{msk}_h}, \mathsf{msk}_h)$, and $(\mathsf{ipk}_h, \mathsf{isk}_h)$, and sends the public keys to the client and the adversary.
3. The client generates random $\{x_j\}_{j \in X_H}$, and $\{c^j = (X_1^j = g^{x_j}, c_1^j)\}_{j \in X_H}$ using the protocol described in §6.2. It also generates a NIZK that it knows x_j that corresponds to X_1^j for each j, and sends both $\{c^j\}$ and the NIZKs to the adversary and the verifier.
4. The adversary generates its input messages $\{c^j = (X_1^j, c_1^j)\}_{j \in X_A}$ (not necessarily by following the protocol in §6.2). It generates a NIZK that shows it knows the discrete log of X_1^j and sends $\{c^j\}_{j \in X_A}$ and the NIZKs to the client and the verifier.
5. The verifier verifies all NIZKs.
6. For $i = 1, \ldots, h-1$, the adversary sends the verifier $\{X_{i+1}^j\}_{j \in [M]}$, and a NIZK that shows

$$\left(\prod_{j=1}^{M} X_i^j\right)^{\mathsf{bsk}_i} = \prod_{j=1}^{M} X_{i+1}^j$$

   by proving that

$$\log_{\prod_{j=1}^{M} X_i^j}\left(\prod_{j=1}^{M} X_{i+1}^j\right) = \log_{\mathsf{bpk}_{i-1}}(\mathsf{bpk}_i)\ .$$

   It also sends the ciphertexts $\{c_h^{j\prime}\}$ to the verifier.
7. The verifier verifies all NIZKs, and checks that $\mathsf{ADec}((X_h^j)^{\mathsf{msk}_h}, c_h^{j\prime}) = (1, \cdot)$ for all $j \in [M]$.

The game halts if the verifier fails to verify any NIZKs or authenticated decryption ever fails (i.e., returns $(0,\cdot)$). The adversary wins the game if the game does not halt and it has successfully tampered with some messages. In other words, the adversary wins if

1. $\left(\prod_{j=1}^{M} X_i^j\right)^{\mathsf{bsk}_i} = \prod_{j=1}^{M} X_{i+1}^j$ for all $i = 1, \ldots, h-1$,
2. there exists $X_T \subset X_H$ such that $|X_T| > 0$ and for all $j \in X_T$ one of the two properties is true: (1) $(X_1^j)^{\prod_{i<h} \mathsf{bsk}_i} \neq X_h^j$, or (2) $(X_1^j)^{\prod_{i<h} \mathsf{bsk}_i} = X_h^j$ and $c_h^j \neq c_h^{j\prime}$ (i.e., the adversary tampered with some messages),
3. and $\mathsf{ADec}((X_h^j)^{\mathsf{msk}_h}, c_h^{j\prime}) = (1, \cdot)$ for all $j \in [M]$.

We will now show that if the adversary can win this game, then it can also break Diffie-Hellman. Assume the adversary won the game. Let $\mathsf{bsk}_A = \prod_{i<h} \mathsf{bsk}_i$ be the product of the private blinding key of the adversary. If the adversary won, then the first condition implies that $(\prod_{j=1}^{M} X_1^j)^{\mathsf{bsk}_A} = \prod_{j=1}^{M} X_h^j$. Now, consider three boolean predicates for j: $c_h^j \overset{?}{=} c_h^{j\prime}$, $(X_1^j)^{\mathsf{bsk}_A} \overset{?}{=} X_h^j$, and $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h})$, where $\mathsf{KNOW}(x) = 1$ if the adversary knows (or can compute) x, and 0 otherwise. There are eight possible combinations of the predicates, and we consider

Table 1: Summary of notation used in aggregate hybrid shuffle.

Notation	Description
M	The total number of users.
N	The total number of servers.
n	The total number of chains.
$(\mathsf{isk}_i, \mathsf{ipk}_i = g^{\mathsf{isk}_i})$	Per-server private-public key pair used to encrypt and decrypt the inner most layer of ciphertexts. Used to protect the messages under active attacks by malicious servers or users.
$(\mathsf{bsk}_i, \mathsf{bpk}_i)$	Per-server private-public key pair used to blind the users' Diffie-Hellman keys. After shuffling and decrypting the messages, each server blinds the Diffie-Hellman keys by raising them to the private key bsk_i. This hides which output message is associated with which input while preserving some structures of the Diffie-Hellman keys, which allows us to generate efficient zero-knowledge proofs to prove the correctness of the shuffle. The public component is used to derive the public keys used to encrypt messages for the mix chain (i.e., $\{\mathsf{mpk}_i\}$). bpk_i is $g^{\prod_{k\leq i}\mathsf{bsk}_k}$ for server i in a mix chain.
$(\mathsf{msk}_i, \mathsf{mpk}_i)$	Per-server private-public key pair used to encrypt and decrypt messages for shuffling. mpk_i is $(g^{\prod_{k<i}\mathsf{bsk}_k})^{\mathsf{msk}_i}$ for server i in a mix chain.
x^j	The private Diffie-Hellman key of user j used to encrypt the messages for a mix chain. The user uses the same x^j for all layers of encryption, but the effective private Diffie-Hellman key of a layer changes after a server processes the messages due to blinding.
X_i^j	The public Diffie-Hellman key of user j used to encrypt her message for server i. If everyone is behaving correctly, then this should be $(g^{\prod_{k<i}\mathsf{bsk}_k})^{x^j}$.
c_i^j	The actual ciphertext component that encrypts the message. User j will derive the encryption key using its private key x^j and the public mixing key mpk_i. Server i will derive the decryption key using its private mixing key msk_i and X_i^j.

each combination for $j \in X_H$. We indicate which combinations are possible for the adversary to satisfy, given that all authenticated decryptions were successful.

1. $c_h^j \neq c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 0$: **IMPOSSIBLE.** Since the adversary does not know the key used to decrypt (i.e., $(X_h^j)^{\mathsf{msk}_h}$), it cannot generate a valid ciphertext.

2. $c_h^j \neq c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1$: **POSSIBLE.** Since the adversary knows the key used to decrypt it could generate a valid ciphertext.

3. $c_h^j \neq c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 0$: **IMPOSSIBLE.** Same argument as case 1.

4. $c_h^j \neq c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1$: **IMPOSSIBLE.** If possible, then the adversary can break the Diffie-Hellman assumption. Namely, given only $X_1^j = g^{x_j}$, bsk_A, and $(g^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$ for random x_j and msk_h and an independently generated bsk_A, it can compute $(X_h^j)^{\mathsf{msk}_h} = g^{x_j \cdot \mathsf{bsk}_A \cdot \mathsf{msk}_h}$. If this were possible, then given g^a and g^b for random a and b, the adversary could generate an bsk_A, compute $(g^b)^{\mathsf{bsk}_A}$, and compute $g^{a \cdot b \cdot \mathsf{bsk}_A}$. It could then break the Diffie-Hellman assumption and compute g^{ab} by raising $g^{a \cdot b \cdot \mathsf{bsk}_A}$ to bsk_A^{-1}.

5. $c_h^j = c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 0$: **IMPOSSIBLE.** If possible, then c_h^j authenticates under two different keys $(X_1^j)^{\mathsf{bsk}_A \cdot \mathsf{msk}_h}$ and $(X_h^j)^{\mathsf{msk}_h}$. However, if we model the underlying hash function of HMAC as a random oracle, then the HMAC instantiated with a random key is a random function. This implies that the probability that the same ciphertext authenticates under two different keys (i.e., collision of two random functions) is negligible when using Encrypt-then-HMAC.

6. $c_h^j = c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1$: **POSSIBLE.** Since the adversary knows and controls the key used for decryption (in particular, the HMAC), it may be able to find a key that is different from the key used to encrypt that results in valid decryption.

7. $c_h^j = c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 0$: **POSSIBLE.** This corresponds to an untampered message.

8. $c_h^j = c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1$: **IMPOSSIBLE.** Same argument as case 4.

Thus, there are three possible combinations of predicates: ($c_h^j \neq c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1$), ($c_h^j = c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} \neq X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 1$), and ($c_h^j = c_h^{j'}$, $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$, $\mathsf{KNOW}((X_h^j)^{\mathsf{msk}_h}) = 0$). The first two cases correspond to X_T (tampered messages), and the second corresponds exactly to $X_H \setminus X_T$ (untampered messages).

Similarly, consider $j \in X_A$. The adversary generates the ciphertexts $\{c_h^{j'}\}$ for the verifier. Thus, the adversary must know $(X_h^j)^{\mathsf{msk}_h}$, the key used to authenticate the ciphertext, for $j \in X_A$.

Now, we consider the product of the users' Diffie-Hellman keys. Because all NIZKs have to be verified, we have that

$$\left(\prod_{j=1}^{M} X_1^j\right)^{\mathsf{bsk}_A} = \left(\prod_{j=1}^{M} X_h^j\right).$$

Consider $X_U = X_H \setminus X_T$, i.e., the set of messages that did not change. Then, we can divide both sides by the values associated with X_U since $(X_1^j)^{\mathsf{bsk}_A} = X_h^j$ for $j \in X_U$:

$$\left(\prod_{j \in X_T \cup X_A} X_1^j\right)^{\mathsf{bsk}_A} = \left(\prod_{j \in X_T \cup X_A} X_h^j\right),$$

since $X_T \cup X_A = [M] \setminus X_U$. We can rewrite this as

$$\left(\prod_{j \in X_T} X_1^j\right)^{\mathsf{bsk}_A} = \left(\prod_{j \in X_T \cup X_A} X_h^j\right) / \left(\prod_{j \in X_A} X_1^j\right)^{\mathsf{bsk}_A}. \quad (1)$$

Based on our analysis of the possible predicates for X_T and X_A, the adversary must know $(X_h^j)^{\mathsf{msk}_h}$ for $j \in X_T \cup X_A$. Moreover, the adversary knows $\log_g(X_1^j)$ for $j \in X_A$ (it was required to prove the knowledge in step 4 of the game). Thus, the adversary can compute $((X_1^j)^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$ by computing $((g^{\mathsf{bsk}_A})^{\mathsf{msk}_h})^{\log_g(X_1^j)}$ for $j \in X_A$ (it knows $\mathsf{mpk}_h = (g^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$). As a result, it can compute

$$\left(\prod_{j \in X_T \cup X_A} \left(X_h^j\right)^{\mathsf{msk}_h}\right) / \prod_{j \in X_A} \left(\left(X_1^j\right)^{\mathsf{bsk}_A}\right)^{\mathsf{msk}_h}.$$

This, however, is

$$\begin{aligned}
&\left(\prod_{j \in X_T \cup X_A} \left(X_h^j\right)^{\mathsf{msk}_h}\right) / \prod_{j \in X_A} \left(\left(X_1^j\right)^{\mathsf{bsk}_A}\right)^{\mathsf{msk}_h} \\
&= \left(\left(\prod_{j \in X_T \cup X_A} X_h^j\right) / \left(\prod_{j \in X_A} X_1^j\right)^{\mathsf{bsk}_A}\right)^{\mathsf{msk}_h} \\
&= \left(\left(\prod_{j \in X_T} X_1^j\right)^{\mathsf{bsk}_A}\right)^{\mathsf{msk}_h},
\end{aligned}$$

where the last step uses the equality from equation 1. This means that given $\{X_1^j = g^{x_j}\}$, bsk_A, and $(g^{\mathsf{bsk}_A})^{\mathsf{msk}_h}$ for unknown random $\{x_j\}$ and msk_h, and bsk_A that was generated independently of $\{x_j\}$ and msk_h, the adversary was able to compute $g^{\mathsf{bsk}_A \cdot \mathsf{msk}_h \cdot \sum_{j \in X_T} x_j}$. This essentially breaks Diffie-Hellman.

In more detail, consider the following adversary A_{DH} that tries to break Diffie-Hellman. A_{DH} is given g^a and g^b for random a and b, and is asked to compute g^{ab}. To compute this, A_{DH} simulates the above game by simulating the clients and the honest server. To do so, A_{DH} chooses the blinding keys $\{\mathsf{bsk}_i\}$, and sets $\mathsf{mpk}_h = (\prod_i g^{\mathsf{bsk}_i})^b = (g^b)^{\mathsf{bsk}_A}$; i.e., the private mixing key of the honest server is b. Then, it generates the inputs from the client by generating a random r_j for $j \in X_H$ and setting $g^{x_j} = g^{a+r_j}$. It also generates the inputs associated with each message. The adversary (a simulator) can generate valid NIZKs of knowledge of discrete logs in the random oracle model [48]. For the first layer of encryption (which is for the "honest server" whose secret key is the unknown b), generate random values for c_h^j, since ciphertexts are indistinguishable from random values. For the other layers of encryption, it can generate correctly authenticated ciphertexts by using g^{x_j}, $\{\mathsf{bsk}_i\}_{i<h}$, and $\{\mathsf{msk}_i\}_{i<h}$ without knowing the value of a.

At the end of the game, the adversary can compute $g^{\mathsf{bsk}_A \cdot b \cdot (\sum_{j \in X_T} x_j)}$, as described previously. From this, the adversary can compute the following:

$$\begin{aligned}
g^{\mathsf{bsk}_A \cdot b \cdot (\sum_{j \in X_T} x_j)} &= g^{\mathsf{bsk}_A \cdot b \cdot (\sum_{j \in X_T} (a+r_j))} \\
&= g^{\mathsf{bsk}_A \cdot b \cdot (|X_T| a + \sum_{j \in X_T} r_j)}.
\end{aligned}$$

Since the adversary knows bsk_A and r_j for $j \in X_T$, it can compute

$$g^{|X_T|ab} = (g^{\mathsf{bsk}_A \cdot b \cdot (|X_T| a + \sum_{j \in X_T} r_j)} / (g^b)^{\mathsf{bsk}_A \sum_{j \in X_T} r_j})^{\mathsf{bsk}_A^{-1}}.$$

From this, it can recover g^{ab}, and break Diffie-Hellman. Therefore, the adversary cannot win the above game if Diffie-Hellman is hard, meaning that it could satisfy at most two out of the three conditions to win the game. In turn, this implies that the honest server will always catch an upstream malicious server misbehaving.

B Security game and proof sketches

We define the security of our system using the following security game played between a challenger and an adversary. Both the challenger and the adversary are given the set of users $[M] = \{1, 2, \ldots, M\}$, the set of servers $[N]$, the fraction f of servers the adversary can compromise, and the number of chains n.

1. The adversary selects the set of malicious servers $A_s \subset [N]$ such that $|A_s| \le f \cdot N$, and the set of malicious users $A_c \subset [M]$ such that $|A_c| \le M - 2$. The adversary sends A_s and A_c to the challenger. Let $H_s = [M] \setminus A_s$ and $H_c = [N] \setminus A_c$ denote the set of honest servers and users.

2. The challenger computes the size of each chain k as a function of f and n, as described in §5.2. Then, it creates n mix chains by repeatedly sampling k servers per group at random. The challenger sends the chain configurations to the adversary.

3. The adversary and the challenger generate blinding keys, mixing keys, and inner keys as described in §6.1 and Appendix A.

4. The adversary picks some honest users $H_t \subset H_c$ such that $|H_t| \geq 2$. It generates sets of chains $\{C_x\}$ for $x \in H_t$ such that $C_x \cap C_y \neq \emptyset$ for all $x, y \in H_t$. For $\mathsf{c} \in [n]$, let $U_\mathsf{c} = \{x \in H_t : \mathsf{c} \in C_x\}$. For each chain c, it also generates the potential messages $\{m^\mathsf{c}_{xy}\}$ for $x, y \in U_\mathsf{c}$ where m^c_{xy} is the message that may be sent from user x to user y in chain c. The adversary sends H_t, $\{\{m^\mathsf{c}_{xy}\}\}$, and $\{C_x\}$ to the challenger.
5. The challenger first verifies that every pair of C_x and C_y intersects at least once. If this is not the case, the game halts. The challenger then performs the following for each chain $\mathsf{c} \in [n]$. First, it creates conversation pairs for each chain c $\{(X_i, Y_i)_\mathsf{c}\} \subset U_\mathsf{c} \times U_\mathsf{c}$ at random such that every $x \in U_\mathsf{c}$ appears in exactly one of the pairs. In other words, every user has a unique conversation partner per chain. (If $X_i = Y_i$, then that user is talking with herself.) For each $(x, y) \in \{(X_i, Y_i)_\mathsf{c}\}$, the challenger onion-encrypts the messages m^c_{xy} from x to y and m^c_{yx} from y to x with the keys of servers in chain c. Then, it uses the protocol described in §6.2 to submit the ciphertexts and the necessary NIZKs.
6. The adversary generates inputs to the chains for the users in A_c, and sends them to the chains.
7. The challenger and the adversary take turns processing the messages in each chain. Within a chain, they perform the following for $i = 1, \ldots, k$:
   (a) If server $i \in H_s$, the challenger performs protocol described in §6.3 to shuffle and decrypt the messages, and also generate an AHS proof. The challenger then sends the proof to the adversary, and the resulting messages to the owner of server $i+1$.
   (b) If server $i \in A_s$, the adversary generates some messages along with an AHS proof. Then, sends the AHS proof to the challenger, and sends the messages to the owner of server $i+1$.

   The challenger verifies all AHS proofs.
8. The challenger and the adversary decrypt the final result of the shuffle (i.e., the inner ciphertexts).
9. The challenger samples a random bit $b \leftarrow 0, 1$. If $b = 0$, then send the adversary $\{(X_i, Y_i)_\mathsf{c}\}$ for $\mathsf{c} \in [n]$. If $b = 1$, then sample random conversation pairs $\{(X'_i, Y'_i)_\mathsf{c}\} \subset U_\mathsf{c} \times U_\mathsf{c}$ for each chain with the same constraint as in step 5, and send the adversary the newly sampled pairs.
10. The adversary makes a guess b' for b.

The adversary wins the game if the game does not come to a halt before the last step and $b' = b$. The adversary need not follow the protocol described in this paper. The advantage of the adversary in this game is $|\Pr[b' = b] - \frac{1}{2}|$. We say that the system provides metadata private communication if the advantage is negligible in the implicit security parameter.

Note that this game models a stronger version of XRD, which allows users to communicate with multiple users on different chains rather than only one user. We could change the game slightly to force the challenger to send loopback messages in step 5 to model having just one conversation.

Proof sketches. First, we argue that the adversary needs to tamper with messages prior to the last honest server shuffling, as stated in §6. To see why, consider an adversary that only tampers with the messages after the last honest server. The adversary can learn the recipients of all messages, but not the senders. As a result, the adversary does not learn anything about whether two users $x, y \in U_\mathsf{c}$ received messages because there exists a conversation pair $(x, y)_\mathsf{c}$, or because there were two conversation pairs $(x, x)_\mathsf{c}$ and $(y, y)_\mathsf{c}$. This means that any set of conversation pairs is equally likely to be sampled by the challenger from the adversary's view, meaning that the adversary does not gain any advantage. Thus, we consider an adversary who tampers with messages prior to the honest server processing the messages.

In this scenario, the adversary in step 7 must follow the protocol (e.g., no tampering with the messages), as analyzed in Appendix A. Given this restriction, we now argue that the adversary does not learn anything after playing the security game by describing how a simulator of an adversary could simulate the whole game with only the public inputs and the private values of the adversary.

The simulator can simulate step 3 by generating random public keys. It can simulate step 5 by generating random values in place of the ciphertexts that encrypt the users' messages $\{\{m^\mathsf{c}_{xy}\}\}$, since the ciphertexts are indistinguishable from random. It then randomly matches a user in U_c to one of the generated random values for each chain c, and sets the destination of each message as the matched user. It onion-encrypts the final message using the randomly generated public keys and the adversary's public keys. In step 7, the adversary simulates the challenger by randomly permuting the messages, and removing a layer of the encryption from the messages. (It can remove a layer of encryption since it knows all layers of onion-encryption.) Finally, it could simulate the challenger's last challenge by picking sets of randomly generated conversation pairs, subject to the constraints in step 5. The distribution of the messages generated and exchanged in the security game and in the simulator are indistinguishable for a computationally limited adversary.

High Throughput Cryptocurrency Routing in Payment Channel Networks

Vibhaalakshmi Sivaraman[*], Shaileshh Bojja Venkatakrishnan[**], Kathleen Ruan[†], Parimarjan Negi[*], Lei Yang [*], Radhika Mittal [‡], Giulia Fanti[†], Mohammad Alizadeh[*]

[*]Massachusetts Insititute of Technology, [**] Ohio State University, [†] Carnegie Mellon University, [‡]University of Illinois at Urbana-Champaign

Abstract

Despite growing adoption of cryptocurrencies, making fast payments at scale remains a challenge. Payment channel networks (PCNs) such as the Lightning Network have emerged as a viable scaling solution. However, completing payments on PCNs is challenging: payments must be routed on paths with sufficient funds. As payments flow over a single channel (link) in the same direction, the channel eventually becomes depleted and cannot support further payments in that direction; hence, naive routing schemes like shortest-path routing can deplete key payment channels and paralyze the system. Today's PCNs also route payments atomically, worsening the problem. In this paper, we present Spider, a routing solution that "packetizes" transactions and uses a multi-path transport protocol to achieve high-throughput routing in PCNs. Packetization allows Spider to complete even large transactions on low-capacity payment channels over time, while the multi-path congestion control protocol ensures balanced utilization of channels and fairness across flows. Extensive simulations comparing Spider with state-of-the-art approaches shows that Spider requires less than 25% of the funds to successfully route over 95% of transactions on balanced traffic demands, and offloads 4x more transactions onto the PCN on imbalanced demands.

1 Introduction

Despite their growing adoption, cryptocurrencies suffer from poor scalability. For example, the Bitcoin [5] network processes 7 transactions per second, and Ethereum [14] 15 transactions/second, which pales in comparison to the 1,700 transactions per second achieved by the VISA network [56]. Scalability thus remains a major hurdle to the adoption of cryptocurrencies for retail and other large-scale applications. The root of the scalability challenge is the inefficiency of the underlying consensus protocol: every transaction must go through full consensus to be confirmed, which can take anywhere from several minutes to hours [43].

A leading proposal among many solutions to improve cryptocurrency scalability [23, 32, 40] relies on so-called *payment channels*. A payment channel is a cryptocurrency transaction that escrows or dedicates money on the blockchain for exchange with a prespecified user for a predetermined duration. For example, Alice can set up a payment channel with Bob in which she escrows 10 tokens for a month. Now Alice can send Bob (and only Bob) signed transactions from the escrow account, and Bob can validate them privately in a secure manner without mediation on the blockchain (§2). If Bob or Alice want to close the payment channel at any point, they can broadcast the most recent signed transaction message to the blockchain to finalize the transfer of funds.

The versatility of payment channels stems from **payment channel networks** (PCNs), in which users who do not share direct payment channels can route transactions through intermediaries for a nominal fee. PCNs enable fast, secure transactions without requiring consensus on the blockchain for every transaction. PCNs have received a great deal of attention in recent years, and many blockchains are looking to PCNs to scale throughput without overhauling the underlying consensus protocol. For example, Bitcoin has deployed the Lightning network [10, 15], and Ethereum uses Raiden [18].

For PCNs to be economically viable, the network must be able to support high *transaction throughput*. This is necessary for intermediary nodes (routers) to profitably offset the opportunity cost of escrowing funds in payment channels, and for encouraging end-user adoption by providing an appealing quality of payment service. But, a transaction is successful only if all channels along its route have sufficient funds. This makes payment channel *routing*, the protocol by which a path is chosen for a transaction, of paramount importance.

Existing payment channel routing protocols achieve poor throughput, for two main reasons. First, they attempt to route each incoming transaction atomically and instantaneously, in full. This approach is harmful, particularly for larger transactions, because a transaction fails completely if there is no path to the destination with enough funds. Second, existing routing protocols fail to keep payment channels *balanced*. A payment channel becomes imbalanced when the transaction rate across it is higher in one direction than the other; the party making more transactions eventually runs out of funds and cannot send further payments without "refilling" the channel via either an on-chain transaction (i.e., committing a new transaction to the blockchain) or coordinated cyclic payments between a series of PCN nodes [39]. Most PCNs today route transactions naively on shortest paths with no consideration for channel balance; this can leave many channels depleted, reducing throughput for everyone in the network. We describe a third problem, the creation of *deadlocks* in certain scenarios, in §3.

In this paper we present *Spider*, a multi-path transport protocol that achieves balanced, high-throughput routing in PCNs, building on concepts in an earlier position paper [51]. Spider's design centers on two ideas that distinguish it from existing approaches. First, Spider senders "packetize" transactions, splitting them into transaction-units that can be sent across different paths at different rates. By enabling congestion-control-like mechanisms for PCNs, this packet-switched approach makes it possible to send large payments on low-capacity payment channels over a period of time. Second, Spider develops a simple multi-path congestion control algorithm that promotes balanced channels while maximizing throughput. Spider's senders use a simple one-bit congestion signal from the routers to adjust window sizes, or the number of outstanding transaction-units, on each of their paths.

Spider's congestion control algorithm is similar to multi-path congestion control protocols like MPTCP [59] developed for Internet congestion control. But the routing problem it solves in PCNs differs from standard networks in crucial ways. Payment channels can only route transactions by moving a finite amount of funds from one end of the channel to the other. Because of this, the capacity of a payment channel — the transaction rate that it can support — varies depending on how it is used; a channel with balanced demand for routing transactions in both directions can support a higher rate than an imbalanced one. Surprisingly, we find that a simple congestion control protocol can achieve such balanced routing, despite not being designed for that purpose explicitly.

We make the following contributions:

1. We articulate challenges for high-throughput routing in payment channel networks (§3), and we formalize the balanced routing problem (§5). We show that the maximum throughput achievable in a PCN depends on the nature of the transaction pattern: circulation demands (participants send on average as much as they receive) can be routed entirely with sufficient network capacity, while demands that form Directed Acyclic Graphs (DAGs) where some participants send more than they receive cannot be routed entirely in a balanced manner. We also show that introducing DAG demands can create deadlocks that stall all payments.
2. We propose a packet-switched architecture for PCNs (§4) that splits transactions into transaction-units and multiplexes them across paths and time.
3. We design Spider (§6), a multi-path transport protocol that (i) maintains balanced channels in the PCN, (ii) uses the funds escrowed in a PCN efficiently to achieve high throughput, and (iii) is fair to different payments.
4. We build a packet-level simulator for PCNs and validate it with a small-scale implementation of Spider on the LND Lightning Network codebase [15]. Our evaluations (§7) show that (i) on circulation demands where 100% throughput is achievable, compared to the state-of-the-art, Spider requires 25% of the funds to route over 95% of the transactions and completes 1.3-1.8x more of the largest 25% of transactions based on a credit card transactions dataset [34]; (ii) on DAG demands where 100% throughput is not achievable, Spider offloads 7-8x as many transactions onto the PCN for every transaction on the blockchain, a 4x improvement over current approaches.

2 Background

Bidirectional payment channels are the building blocks of a payment channel network. A bidirectional payment channel allows a sender (Alice) to send funds to a receiver (Bob) and vice versa. To open a payment channel, Alice and Bob jointly create a transaction that escrows money for a fixed amount of time [46]. Suppose Alice puts 3 units in the channel, and Bob puts 4 (Fig. 1). Now, if Bob wants to transfer one token to Alice, he sends her a cryptographically-signed message asserting that he approves the new balance. This message is not committed to the blockchain; Alice simply holds on to it. Later, if Alice wants to send two tokens to Bob, she sends a signed message to Bob approving the new balance (bottom left, Fig. 1). This continues until one party decides to close the channel, at which point they publish the latest message to the blockchain asserting the channel balance. If one party tries to cheat by publishing an earlier balance, the cheating party loses all the money they escrowed to the other party [46].

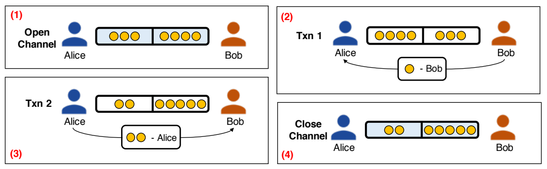


Figure 1: Bidirectional payment channel between Alice and Bob. A blue shaded block indicates a transaction that is committed to the blockchain.

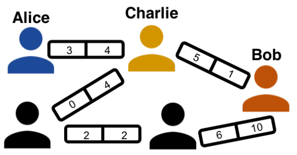


Figure 2: In a payment channel network, Alice can transfer money to Bob by using intermediate nodes' channels as relays. There are two paths from Alice to Bob, but only the path (Alice, Charlie, Bob) can support 3 tokens.

A payment channel network is a collection of bidirectional payment channels (Fig. 2). If Alice wants to send three tokens to Bob, she first finds a path to Bob that can support three tokens of payment. Intermediate nodes on the path (Charlie) will relay payments to their destination. Hence in Fig. 2, two transactions occur: Alice to Charlie, and Charlie to Bob. To incentivize Charlie to participate, he receives a routing fee. To prevent him from stealing funds, a cryptographic hash lock ensures that all intermediate transactions are only valid after a transaction recipient knows a private key generated by Alice [18]. [1] Once Alice is ready to pay, she gives that key to

[1]The protocol called Hashed Timelock Contracts (HTLCs) can be implemented in two ways: the sender generates the key, as in Raiden [18] or the receiver generates the key, as in Lightning [46]. Spider assumes that the sender generates the key.

Bob out-of-band; he can either broadcast it (if he decides to close the channel) or pass it to Charlie. Charlie is incentivized to relay the key upstream to Alice so that he can also get paid. Note that Charlie's payment channels with Alice and Bob are independent: Charlie cannot move funds between them without going through the blockchain.

3 Challenges in Payment Channel Networks

A major cost of running PCNs is the collateral needed to set up payment channels. As long as a channel is open, that collateral is locked up, incurring an opportunity cost for the owner. For PCNs to be financially viable, this opportunity cost should be offset by routing fees, which are charged on each transaction that passes through a router. To collect more routing fees, routers try to process as many transactions as possible for a given amount of collateral. A key performance metric is therefore the *transaction throughput per unit collateral* where throughput itself is measured either in number of transactions per second or transaction value per second.

Current PCN designs exhibit poor throughput due to naive design choices in three main areas: (1) *how* to route transactions,(2) *when* to send them and, (3) *deadlocks*.

Challenge #1: How to route transactions? A central question in PCNs is what route(s) to use for sending a transaction from sender to destination. PCNs like the Lightning and Raiden networks are source-routed. [2] Most clients by default pick the shortest path from the source to the destination.

However, shortest-path routing degrades throughput in two key ways. The first is to cause underutilization of the network. To see this, consider the PCN shown in Fig. 3a. Suppose we have two clusters of nodes that seek to transact with each other at roughly the same rate on average, and the clusters are connected by two paths, one consisting of channels $a-b$, and the other channel c. If the nodes in cluster A try to reach cluster B via the shortest path, they would all take channel c, as would the traffic in the opposite direction. This leads to congestion on channel c, while channels a and b are under-utilized.

A second problem is more unique to PCNs. Consider a similar topology in Figure 3b, and suppose we fully utilize the network by sending all traffic from cluster A→B on edge a and all traffic from cluster B→A on edge b. While the rate on both edges is the same, as funds flow in one direction over a channel, the channel becomes *imbalanced*: all of the funds end up on one side of the channel. Cluster A can no longer send payments until it receives funds from cluster B on the edge a or it deposits new funds into the channel a via an on-chain transaction. The same applies to cluster B on edge b. Since on-chain transactions are expensive and slow, it is desirable to avoid them. Routing schemes like shortest-path routing do not account for this problem, thereby leading to reduced throughput (§7). In contrast, it is important to choose routes that actively prevent channel imbalance. For example, in Figure 3b, we could send half of the A→B traffic on edge a, and half on edge b, and the same for the B→A traffic. The challenge is making these decisions in a fully decentralized way.

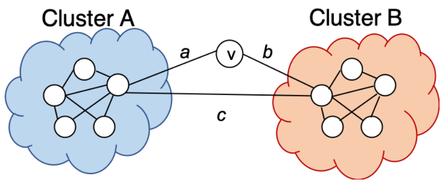
Challenge #2: When to send transactions? Another problem is *when* to send transactions. Most existing PCNs are circuit-switched: transactions are processed instantaneously and atomically upon arrival [18, 46]. This causes a number of problems. If a transaction's value exceeds the available balance on each path from the source to the destination, the transaction fails. Since transaction values in the wild tend to be heavy-tailed [29, 34], either a substantial fraction of real transactions will fail as PCN usage grows, or payment channel operators will need to provision higher collateral to satisfy demand.

Even when transactions do not fail outright, sending transactions instantaneously and atomically exacerbates the imbalance problem by transferring the full transaction value to one side of the channel. A natural idea to alleviate these problems is to "packetize" transactions: transactions can be split into smaller transaction-units that can be multiplexed over space (by traversing different paths) and in time (by being sent at different rates). Versions of this idea have been proposed before; atomic multi-path payments (AMP) enable transactions to traverse different paths in the Lightning network [3], and the Interledger protocol uses a similar packetization to conduct cross-ledger payments [54]. However, a key observation is that it is not enough to subdivide transactions into smaller units: to achieve good throughput, it is also important to multiplex in *time* as well, by performing congestion control. If there is a large transaction in one direction on a channel, simply sending it out in smaller units that must all complete together doesn't improve the likelihood of success. Instead, in our design, we allow each transaction-unit to complete independently, and a congestion control algorithm at the sender throttles the rate of these units to match the rate of units in the opposite direction at the bottlenecked payment channel. This effectively allows the tokens at that bottleneck to be replenished and reused multiple times as part of the same transaction, achieving a multiplicative increase in throughput for the same collateral.

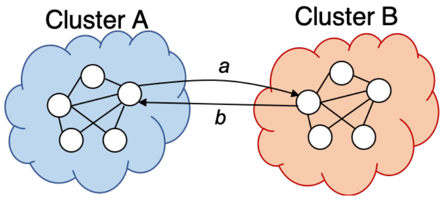
Challenge #3: Deadlocks. The third challenge in PCNs is the idea that the introduction of certain flows can actively harm the throughput achieved by other flows in the network. To see this, consider the topology and demand rates in Figure 3c. Suppose nodes 1 and 2 want to transmit 1-unit transactions to node 3 at rates of 1 and 2 units/second, respectively, and node 3 wants to transact 2 units/sec with node 1.[3] Notice that the specified transaction rates are imbalanced: there is a net flow of funds out of node 2 and into nodes 1 and 3. Suppose the payment channels are initially balanced, with 10 units on each side and we only start out with flows between nodes 1 and 3. For this demand and topology, the system can sustain 2 units/sec by only having nodes 1 and 3 to send to each other at a rate of 1 unit/second.

[2] This was done in part for privacy reasons: transactions in the Lightning network use onion-routing, which is easy to implement with source routing [33].

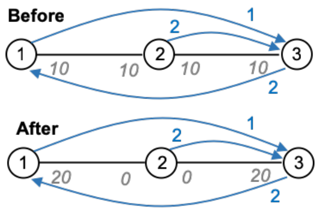
[3] For simplicity, we show three nodes, but a node in this example could represent a cluster of many users who wish to transact at the rates shown in aggregate.



(a) Underutilized channels



(b) Imbalanced channels



(c) Deadlock

Figure 3: Example illustrating the problems with state-of-the-art PCN routing schemes.

However, once transactions from node 2 are introduced, this example achieves zero throughput at steady-state. The reason is that node 2 sends transactions to node 3 faster than its funds are being replenished, which reduces its funds to 0. Slowing down 2's transactions would only delay this outcome. Since node 2 needs a positive balance to route transactions between nodes 1 and 3, the transactions between 1 and 3 cannot be processed, despite the endpoints having sufficient balance. The network finds itself in a *deadlock* that can only be resolved by node 2 replenishing its balance with an on-chain transaction.

Why these problems are difficult to solve. The above problems are challenging because their effects are closely intertwined. For example, because poor routing and rate-control algorithms can cause channel imbalance, which in turn degrades throughput, it is difficult to isolate the effects of each. Similarly, simply replacing circuit switching with packet-switching gives limited benefits without a corresponding rate control and routing mechanism.

From a networking standpoint, PCNs are very different from traditional communication networks: payment channels do not behave like a standard communication link with a certain capacity, say in transactions per second. Instead, the capacity of a channel in a certain direction depends on two factors normally not seen in communication networks: (a) the rate that transactions are received in the reverse direction on that channel, because tokens cannot be sent faster on average in one direction than they arrive in the other, (b) the delay it takes for the destination of a transaction to receive it and send back the secret key unlocking the funds at routers (§2). Tokens that are "in flight", i.e. for which a router is waiting for the key, cannot be used to service new transactions. Therefore the network's capacity depends on its delay, and queued up transactions at a depleted link can hold up funds from channels in other parts of the network. This leads to cascading effects that make congestion control particularly critical.

4 Packet-Switched PCN

Spider uses a packet-switched architecture that splits transactions into a series of independently routed *transaction-units*. Each transaction-unit transfers a small amount of money bounded by a *maximum-transaction-unit (MTU)* value. Packetizing transactions is inspired by packet switching for the Internet, which is more effective than circuit switching [41]. Note that splitting transactions does not compromise the security of payments; each transaction-unit can be created with an independent secret key. As receivers receive and acknowledge transaction-units, senders can selectively reveal secret keys only for acknowledged transaction-units (§2). Senders can also use proposals like Atomic Multi-Path Payments (AMP) [3] if they desire atomicity of transactions.

In Spider, payments transmitted by source *end-hosts* are forwarded to their destination end-hosts by *routers* within the PCN. Spider routers queue up transaction-units at a payment channel whenever the channel lacks the funds to forward them immediately. As a router receives funds from the other side of its payment channel, it uses these funds to forward transaction-units waiting in its queue. Current PCN implementations [15] do not queue transactions at routers—a transaction fails immediately if it encounters a channel with insufficient balance on its route. Thus, currently, even a temporary lack of channel balance can cause many transactions to fail, which Spider avoids.

5 Modeling Routing

A good routing protocol must satisfy the following objectives:

1. **Efficiency.** For a PCN with a fixed amount of escrowed capital, the aggregate transaction throughput achieved must be as high as possible.
2. **Fairness.** The throughput allocations to different users must be fair. Specifically, the system should not starve transactions of some users if there is capacity.

Low latency, a common goal in communication networks, is desirable but not a first order concern, as long as transaction latency on the PCN is significantly less than an on-chain transaction (which can take minutes to hours today). However, as mentioned previously (§3), very high latency could hurt the throughput of a PCN, and must therefore be avoided. We assume that the underlying communication network is not a bottleneck and PCN users can communicate payment attempts, success and failures with one another easily since these messages do not require much bandwidth.

To formalize the routing problem, we consider a fluid model of the system in which payments are modeled as continuous "fluid flows" between users. This allows us to cast routing as an optimization problem and derive decentralized algorithms from it, analogous to the classical Network Utility Maximization (NUM) framework for data networks [45]. More specifically, for the fluid model we consider a PCN modeled as a graph $G(V,E)$ in which V denotes the set of nodes (i.e., end-hosts or routers), and E denotes the set of payment channels between them. For a path p, let x_p denote the (fluid) rate at which payments are sent along p from a source to a destination. The fluid rate captures the long-term average rate at which payments are made on the path.

For maximizing throughput efficiency, routing has to be done such that the total payment flow through each channel is as high as possible. However, routers have limited capital on their payment channels, which restricts the maximum rate at which funds can be routed (Fig. 3a). In particular, when transaction units are sent at a rate $x_{u,v}$ across a payment channel between u and v with $c_{u,v}$ funds in total and it takes Δ time units on average to receive the secret key from a destination once a payment is forwarded, then $x_{u,v}\Delta$ credits are locked (i.e., unavailable for use) at any point in time in the channel. This implies that the average rate of transactions (across both directions) on a payment channel cannot exceed $c_{u,v}/\Delta$. This leads to *capacity constraints* on channels.

Sustaining a flow in one direction through a payment channel requires funds to be regularly replenished from the other direction. This requirement is a key difference between PCNs and traditional data networks. In PCNs if the long-term rates $x_{u,v}$ and $x_{v,u}$ are mismatched on a channel (u,v), say $x_{u,v} > x_{v,u}$, then over time all the funds $c_{u,v}$ will accumulate at v deeming the channel unusable in the direction u to v (Fig. 3b). This leads to *balance constraints* which stipulate that the total rate at which transaction units are sent in one direction along a payment channel matches the total rate in the reverse direction.

Lastly, for enforcing fairness across flows we assume sources have an intrinsic *utility* for making payments, which they seek to maximize. A common model for utility at a source is the logarithm of the total rate at which payments are sent from the source [31, 37, 38]. A logarithmic utility ensures that the rate allocations are proportionally fair [38]—no individual sender's payments can be completely throttled. Maximizing the overall utility across all source-destination pairs subject to the capacity and balance constraints discussed above, can then be computed as

$$\text{maximize} \quad \sum_{i,j\in V} \log\Big(\sum_{p\in\mathcal{P}_{i,j}} x_p\Big) \tag{1}$$

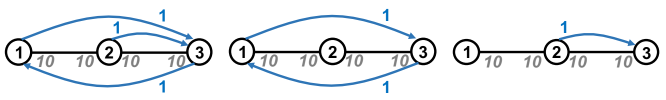
$$\text{s.t.} \quad \sum_{p\in\mathcal{P}_{i,j}} x_p \le d_{i,j} \quad \forall i,j\in V \tag{2}$$

$$x_{u,v}+x_{v,u} \le \frac{c_{u,v}}{\Delta} \quad \forall(u,v)\in E \tag{3}$$

$$x_{u,v} = x_{v,u} \quad \forall(u,v)\in E \tag{4}$$

$$x_p \ge 0 \quad \forall p\in\mathcal{P}, \tag{5}$$

where for a source i and destination j, $\mathcal{P}_{i,j}$ is the set of all paths from i to j, $d_{i,j}$ is the demand from i to j, $x_{u,v}$ is the total flow going from u to v for a channel (u,v), $c_{u,v}$ is the total amount of funds escrowed into (u,v), Δ is the average round-trip time of the network taken for a payment to be completed, and $\mathcal{P}$ is the set of all paths. Equation (2) specifies *demand constraints* which ensures that the total flow for each sender-receiver pair across all of their paths, is no more than their demand.



(a) Payment graph (b) Circulation (c) DAG

Figure 4: Payment graph (denoted by blue lines) for a 3 node network (left). It decomposes into a maximum circulation and DAG components as shown in (b) and (c).

5.1 Implications for Throughput

A consequence of the balance constraints is that certain traffic demands are more efficient to route than certain others. In particular, demands that have a *circulation* structure (total outgoing demand matches total incoming demand at a router) can be routed efficiently. The cyclic structure of such demands enables routing along paths such that the rates are naturally balanced in channels. However, for demands without a circulation structure, *i.e.,* if the demand graph is a directed acyclic graph (DAG), balanced routing is impossible to achieve in the absence of periodic replenishment of channel credits, regardless of how large the channel capacities are.

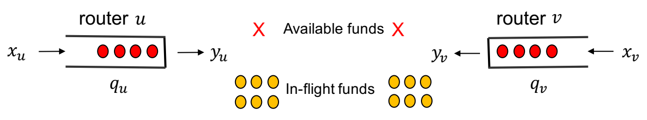
For instance, Fig. 4a shows the traffic demand graph for a PCN with nodes $\{1,2,3\}$ and payment channels between nodes $1-2$ and $2-3$. The weight on each blue edge denotes the demand in transaction-units per second between a pair of users. The underlying black lines denote the topology and channel sizes. Fig. 4b shows the circulation component of the demand in Fig. 4a. The entire demand contained in this circulation can be routed successfully as long as the network has sufficient capacity. In this case, if the confirmation latency for transaction-units between 1 and 3 is less than 10s, then the circulation demand can be satisfied indefinitely. The remaining component of the demand graph, which represents the DAG, is shown in Fig. 4c. This portion cannot be routed indefinitely since it shifts all tokens onto node 3 after which the $2-3$ channel becomes unusable.

App. A formalizes the notion of circulation and shows that the maximum throughput achievable by any balanced routing scheme is at most the total demand contained within the circulation.

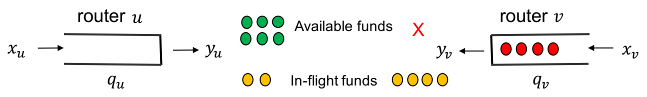
6 Design

6.1 Intuition

Spider routers queue up transactions at a payment channel whenever the channel lacks funds to forward them immediately (§5). Thus, queue buildup is a sign that either transaction-units



(a) A capacity limited payment channel.



(b) An imbalance limited payment channel.

Figure 5: Example of queue growth in a payment channel between routers u and v, under different scenarios of transaction arrival rates at u and v. (a) If the rate of arrival at v, x_v, and the rate of arrival at u, x_u, are such that their sum exceeds the channel capacity, neither router has available funds and queues build up at both u and v. (b) If the arrival rates are imbalanced, *e.g.*, if $x_v > x_u$, then u has excess funds while v has none, causing queue build-up at v.

are arriving faster (in both directions) than the channel can process (Fig. 5a) or that one end of the payment channel lacks sufficient funds(Fig. 5b). It indicates that the capacity constraint (Equation 3) or the balance constraint (Equation 4) is being violated and the sender should adjust its sending rate.

Therefore, if senders use a congestion control protocol that controls queues, they could detect both capacity and imbalance violations and react to them. For example, in Fig. 5a, the protocol would throttle both x_u and x_v. In Fig. 5b, it would decrease x_v to match the rate at which queue q_v drains, which is precisely x_u, the rate at which new funds become available at router v.

This illustrates that a congestion controller that satisfies two basic properties can achieve both efficiency and balanced rates:

1. *Keeping queues non-empty,* which ensures that any available capacity is being utilized, *i.e.,* there are no unused tokens at any router.
2. *Keeping queues stable (bounded),* which ensures that (a) the flow rates do not exceed a channel's capacity, (b) the flow rates are balanced. If either condition is violated, then at least one of the channel's queues would grow.

Congestion control algorithms that satisfy these properties abound (*e.g.,* Reno [19], Cubic [35], DCTCP [22], Vegas [27], etc.) and could be adapted for PCNs.

In PCNs, it is desirable to transmit transaction-units along multiple paths to better utilize available capacity. Consequently, Spider's design is inspired by multi-path transport protocols like MPTCP [59]. These protocols couple rate control decisions for multiple paths to achieve both high throughput and fairness among competing flows [58]. We describe an MPTCP-like protocol for PCNs in §6.2–6.3. In §6.4 we show that the rates found by Spider's protocol for parallel network topologies, match the solution to the optimization problem in §5.

6.2 Spider Router Design

Fig. 6 shows a schematic diagram of the various components in the Spider PCN. Spider routers monitor the time that each

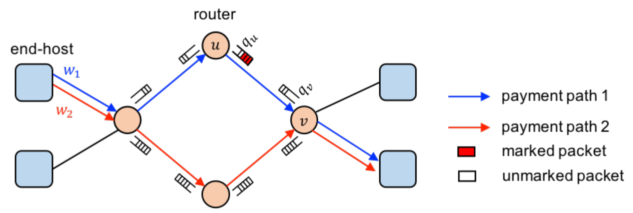


Figure 6: Routers queue up transaction-units and schedule them based on priorities when funds become available. and transaction priorities. If the delay through the queue for a packet exceeds a threshold, they mark the packet. End-hosts maintain and adjust windows for each path to a receiver based on the marks they observe.

packet spends in their queue and mark the packet if the time spent exceeds a pre-determined threshold T. If the transaction-unit is already marked, routers leave the field unchanged and merely forward the transaction-unit. Routers forward acknowledgments from the receiving end-host back to the sender which interprets the marked bit in the ack accordingly. Spider routers schedule transaction-units from their queues according to a scheduling policy, like Smallest-Payment-First or Last-In-First-Out (LIFO). Our evaluations (§7.5) shows that LIFO provides the highest transaction success rate. The idea behind LIFO is to prioritize transaction units from new payments, which are likely to complete within their deadline.

6.3 Spider Transport Layer at End-Hosts

Spider senders send and receive payments on a PCN by interfacing with their transport layer. This layer is configured to support both atomic and non-atomic payments depending on user preferences. Non-atomic payments utilize Spider's packet-switching which breaks up large payments into transaction-units that are delivered to the receiver independently. In this case, senders are notified of how much of the payment was completed allowing them to cancel the rest or retry it on the blockchain. While this approach crucially allows token reuse at bottleneck payment channels for the same transaction (§3), senders also have the option of requesting atomic payments (likely for a higher fee). Our results (§7) show that even with packetization, more than 95% payments complete in full

The transport layer also involves a multi-path protocol which controls the rates at which payments are transferred, based on congestion in the network. For each destination host, a sender chooses a set of k paths to route transaction-units along. The route for a transaction-unit is decided at the sender before transmitting the unit. It is written into the transaction-unit using onion encryption, to hide the full route from intermediate routers [17, 33]. In §7.5, we evaluate the impact of different path choices on Spider's performance and propose using edge-disjoint widest paths [21] between each sender and receiver in Spider.

To control the rate at which payments are sent on a path, end-hosts maintain a window size w_p for every candidate path to a destination. This window size denotes the maximum number of transaction-units that can be outstanding on path p at any point in time. End-hosts track the transaction-units that have been sent out on each path but have not yet been acked or canceled. A new transaction-unit is transmitted on a path p only if the total amount pending does not exceed w_p.

End-hosts adjust w_p based on router feedback on congestion and imbalance. In particular, on a path p between source i and receiver j the window changes as

$$w_p \leftarrow w_p - \beta, \qquad \text{on every marked packet and,} \tag{6}$$

$$w_p \leftarrow w_p + \frac{\alpha}{\sum_{p':p' \in \mathcal{P}_{i,j}} w_{p'}}, \qquad \text{on every unmarked packet.} \tag{7}$$

Here, α and β are both positive constants that denote the aggressiveness with which the window size is increased and decreased respectively. Eq. (6)–(7) are similar to MPTCP, but with a multiplicative decrease factor that depends on the fraction of packets marked on a path (similar to DCTCP [22]).

We expect the application to specify a deadline for every transaction. If the transport layer fails to complete the payment within the deadline, the sender cancels the payment, clearing all of its state from the PCN. In particular, it sends a cancellation message to remove any transaction-units queued at routers on each path to the receiver. Notice that transaction-units that arrive at the receiver in the meantime cannot be unlocked because we assume the sender holds the secret key (§2). Senders can then choose to retry the failed portion of the transaction again on the PCN or on the blockchain; such retries would be treated as new transactions. Canceled packets are considered marked and Spider decreases its window in response to them.

6.4 Optimality of Spider

Under a fluid approximation model for Spider's dynamics, we can show that the rates computed by Spider are an optimal solution to the routing problem in Equations (1)–(5) for parallel networks (such as Fig. 20 in App. B). In the fluid model, we let $x_p(t)$ denote the rate of flow on a path p at time t; for a channel (u,v), $f_{u,v}(t)$ denotes the fraction of packets that are marked at router u as a result of excessive queuing. The dynamics of the flow rates $x_p(t)$ and marking fractions $f_{u,v}(t)$ can be specified using differential equations to approximate the window update dynamics in Equations (6) and (7). We elaborate more on this fluid model, including specifying how the queue sizes and marking fractions evolve, in App. B.

Now, consider the routing optimization problem (Equations (1)–(5)) written in the context of a parallel network. If Spider is used on this network, we can show that there is a mapping from the rates $\{x_p\}$ and marking fractions $\{f_{u,v}\}$ values after convergence, to the primal and dual variables of the optimization problem, such that the Karush-Kuhn-Tucker (KKT) conditions for the optimization problem are satisfied. This proves that the set of rates found by Spider is an optimal solution to the optimization problem [26]. The complete and formal mathematical proof showing the above is presented in App. B.

7 Evaluation

We develop an event-based simulator for PCNs, and use it to extensively evaluate Spider across a wide range of scenarios. We describe our simulation setup (§7.1), validate it via a prototype implementation (§7.2), and present detailed results for circulation demands (§7.3). We then show the effect of adding DAG components to circulations (§7.4), and study Spider's design choices (§7.5).

7.1 Experimental Setup

Simulator. We extend the OMNET++ simulator (v5.4.1) [1] to model a PCN. Our simulator accurately models the network-wide effects of transaction processing, by explicitly passing messages between PCN nodes (endhosts and routers).[4] Each endhost (i) generates transactions destined for other endhosts as per the specified workload, and (ii) determines when to send a transaction and along which path, as per the specified routing scheme. All endhosts maintain a view of the entire PCN topology, to compute suitable source-routes. The endhosts can't view channel balances, but they do know each channel's size or total number of tokens (€). Endhosts also split generated transactions into MTU-sized segments (or transaction-units) before routing, if required by the routing scheme (e.g. by Spider). Each generated transaction has a *timeout* value and is marked as a failure if it fails to reach its destination by then. Upon receiving a transaction, an endhost generates an acknowledgment that is source-routed along its reverse path.

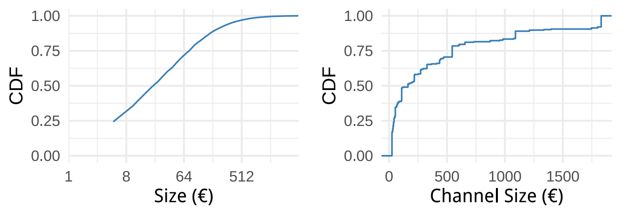
A router forwards incoming transactions and acknowledgments along the payment channels specified in their route, while correspondingly decrementing or incrementing the channel balances. Funds consumed by a transaction in a channel are *inflight* and unavailable until its acknowledgment is received. A transaction is forwarded on a payment channel only if the channel has sufficient balance; otherwise the transaction is stored in a *per-channel queue* that is serviced in a last in first out (LIFO) order §7.5. If the queue is full, an incoming transaction is dropped, and a failure message is sent to the sender.
Routing Schemes. We implement and evaluate five different routing schemes in our simulator.
(1) Spider: Every Spider sender maintains a set of up to k edge-disjoint widest paths to each destination and a window size per path. The sender splits transactions into transaction-units and sends a transaction-unit on a path if the path's window is larger than amount inflight on the path. If a transaction-unit cannot be sent, it is placed in a per-destination queue at the sender that is served in LIFO order. Spider routers mark transaction-units experiencing queuing delays higher than a pre-determined threshold. Spider receivers echo the mark back to senders who adjust the window size according to the equations in §6.3.

[4]https://github.com/spider-pcn/spider-omnet

(2) Waterfilling: Waterfilling uses balance information explicitly in contrast to Spider's 1-bit feedback. As with Spider, a sender splits transactions into transaction-units and picks up to k edge-disjoint widest paths per destination. It maintains one outstanding probe per path that computes the bottleneck (minimum) channel balance along it. When a path's probe is received, the sender computes the available balance based on its bottleneck and the in-flight transaction-units. A transaction-unit is sent along the path with the highest available balance. If the available balance for all of the k paths is zero (or less), the transaction-unit is queued and retried after the next probe.
(3) Shortest Path: This baseline sends transactions along the shortest path to the destination without transaction splitting.
(4) Landmark Routing: Landmark routing, as used in prior PCN routing schemes [42, 47, 50], chooses k well-connected *landmark* nodes in the topology. For every transaction, the sender computes its shortest path to each landmark and concatenates it with the shortest path from that landmark to the destination to obtain k distinct paths. Then, the sender probes each path to obtain its bottleneck balance, and partitions the transaction such that each path can support its share of the total transaction. If such a partition does not exist or if any of the partitions fail, the transaction fails.
(5) LND: The PCN scheme currently deployed in the Lightning Network Daemon (LND) [15] attempts first send a transaction along the shortest path to its destination. If the transaction fails due to insufficient balance at a channel, the sender removes that channel from its local view, recomputes the shortest path, and retries the transaction on the new path until the destination becomes unreachable or the transaction times out. A channel is added back to the local view 5 seconds after its removal.
(6) Celer: App. C.1 compares Spider to Celer's cRoute as proposed in a white-paper [11]. Celer is a back-pressure routing algorithm that routes transactions based on queue and imbalance gradients. Due to computation overheads associated with Celer's large queues, we evaluate it on a smaller topology.
Workload. We generate two forms of payment graphs to specify the rate at which a sender transacts with every other receiver: (i) pure circulations, with a fixed total sending rate x per sender generated by adding x random permutation matrices; (ii) circulations with a DAG component, having a total rate y generated by sampling y different sender-receiver pairs where senders and receivers are chosen from two separate exponential distributions. The distribution's skew is set proportional to the desired DAG component in the total traffic matrix.

We translate the rates from the payment graph to discrete transactions with a Poisson arrival process The transaction size distribution (Fig. 7a) is drawn from credit card transaction data [34], and has a mean of 88€ and median 25€ with the largest transaction being 3930€. Each sender sends 30 tx/sec on average shared across 10 destinations. Note that a sender represents a router in our setup, sending transactions to other routers on behalf of many users.
Topology. We set up an LND node [15] to retrieve the Lightning Network topology on July 15, 2019. We snowball sample [36] the full topology (which has over 5000 nodes and 34000 edges), resulting in a PCN with 106 nodes and 265 payment channels. For compatibility with our transaction dataset, we convert LND payment channel sizes from Satoshis to €, and set the minimum channel size to the median transaction size of 25€. The distribution of channel sizes for this topology has a mean and median size of 421€ and 163€ respectively (Fig. 7b). This distribution is highly skewed, resulting in a mean that is much larger than the median or the smallest payment channels. We refer to this distribution as the Lightning Channel Size Distribution (LCSD). We draw channel propagation delays based on ping times from our LND node to all reachable nodes in the Lightning Network, resulting in RTTs of about a second.

We additionally simulate two synthetic topologies: a Watts-Strogatz small world topology [20] with 50 nodes and 200 edges, and a scale-free Barabasi-Albert graph [4] with 50 nodes and 336 edges. We set the per-hop delay to 30ms in both cases, resulting in RTTs of 200-300ms. For payment channel sizes, we use real capacities in the Lightning topology and sample capacities from LCSD for synthetic topologies. We vary the mean channel size across experiments by proportionally scaling up the size of each payment channel. All payment channels are initialized with perfect balance.
Parameters. We set the MTU as 1€. Every transaction has a timeout of 5 seconds. Schemes with router queues enabled have a per-channel queue size of 12000€. The number of path choices is set to $k = 4$ for schemes that use multiple paths. We vary both the number of paths and the nature of paths in §7.5. For Spider, we set α (window increase factor) to 10, β (multiplicative decrease factor) to 0.1, and the marking threshold for the queue delay to 300ms. For the experiments in §7.4, we set this threshold to 75ms to for faster response to congestion.
Metrics. We use the following evaluation metrics: (i) *transaction success ratio:* the number of completed transactions over the number of generated transactions. A packetized transaction is complete when all of its transaction-units are successful, (ii) *normalized throughput:* the total amount of payments (in €) completed over the total amount of payments generated, (iii) *transaction latency*: time between arrival and completion for successful transactions, and (iv) *offload factor*: number of transactions offloaded to the PCN for every on-chain transaction. All of these metrics are computed over a measurement



(a) Transaction Size Distribution (b) LN Channel Size Distribution

Figure 7: Transaction dataset and channel size distribution used for real-world evaluations.

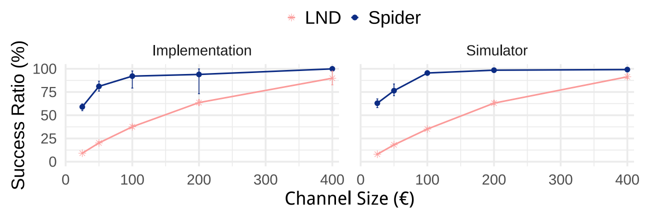


Figure 8: Comparison of performance on simulator and implementation for LND and Spider on a 10 node scale-free topology with 1€ transactions. Spider outperforms LND in both settings. Further, the average success ratio on the simulator and implementation for both schemes are within 5% of each other.

interval when all schemes are in steady-state. Unless specified otherwise, we use a measurement interval of 800-1000s, run experiments for 1010s, and denote the maximum and minimum statistic across five runs using error-bars.

7.2 Prototype Implementation

To support Spider, we modify the Lightning Network Daemon (LND) [15] which is currently deployed on the live Bitcoin Network. We repurpose the router queues to queue up transactions (or HTLCs) that cannot be immediately serviced. When a transaction spends more than 75ms in the queue, Spider marks it. The marking is echoed back via an additional field in the transaction acknowledgement (`FulfillHTLC`) to the sender. We maintain a per-receiver state at the sender to capture the window and number inflight on each path, as well as the queue of unattempted transactions. Each sender finds 4 edge-disjoint shortest paths to every destination. We do not implement transaction-splitting.

We deploy our modified LND implementation [15] on Amazon EC2's `c5d.4xlarge` instances with 16 CPU cores, 16 GB of RAM, 400 GB of NVMe SSD, and a 10 Gbps network interface. Each instance hosts one end-host and one router. Every LND node is run within a docker container with a dedicated bitcoin daemon [6]. We create our own regtest [8] blockchain for the nodes. Channels are created corresponding to a scale-free graph with 10 nodes and 25 edges. We vary the mean channel size from 25€ to 400€. Five circulation payment graphs are generated with each sender sending 100 tx/s (each 1€). Receiving nodes communicate invoices via etcd [13] to sending nodes who then complete them using the appropriate scheme. We run LND and Spider on the implementation and measure the transaction RTTs to inform propagation delays on the simulator. We then run the same experiments on the simulator.

Fig. 8 shows the average success ratio that Spider and LND achieve on the implementation and the simulator. There are two takeaways: (i) Spider outperforms LND in both settings and, (ii) the average success ratio on the simulator is within 5% of the implementation for both schemes. Our attempts at running experiments at larger scale showed that the LND codebase is not optimized for high throughput. For example, persisting HTLC state on disk causes IO bottlenecks and variations of tens of seconds in transaction latencies even on small topologies. Given the fidelity and flexibility of the simulator, we chose to use it for the remaining evaluations.

7.3 Circulation Payment Graph Performance

Recall that on circulation payment graphs, *all* the demand can theoretically be routed if there is sufficient capacity (§5.1 and App. A). However, the capacity at which a routing scheme attains 100% throughput depends on the scheme's ability to balance channels: the more balanced a scheme is, the less capacity it needs for high throughput.

Efficiency of Routing Schemes. We run five circulation traffic matrices on our three topologies (§7.1). Notice that the channel sizes are much larger on the Lightning Topology compared to the other two due to the highly skewed nature of capacities (Fig. 7b). We measure success ratio for the transactions across different channel sizes. Fig. 9 shows that on all topologies, Spider outperforms the state-of-the-art schemes. Spider successfully routes more than 95% of the transactions with less than 25% of the capacity required by LND. At lower capacities, Spider completes 2-3$\times$ more transactions than LND. This is because Spider maintains balance in the network by responding quickly to queue buildup at payment channels, thus making better use of network capacity. The explicit balance-aware scheme, Waterfilling, also routes more transactions than LND. However, when operating in low capacity regimes, where many paths are congested and have near-zero available balance, senders are unable to use just balance information to differentiate paths. As a result, Waterfilling's performance degrades at low capacity compared to Spider which takes into account queuing delays.

Size of Successful Payments. Spider's benefits are most pronounced at larger transaction sizes, where packetization and congestion control helps more transactions complete. Fig. 10 shows success ratio as a function of transaction size. We use mean channel sizes of 4000€ and 16880 € for the synthetic and real topologies, respectively. Each shaded region denotes a different range of transaction sizes, each corresponding to about 12.5% of the transactions in the workload. A point within a range represents the average success ratio for transactions in that interval across 5 runs. Spider outperforms LND across all sizes, and is able to route 5-30% more of the largest transactions compared to LND.

Impact on Latency. We anticipate Spider's rate control mechanism to increase latency. Fig. 11 shows the average and 99th percentile latency for successful transactions on the Lightning topology as a function of transaction size. Spider's average and tail latency increase with transaction size because larger transactions are multiplexed over longer periods of time. However, the tail latency increases much more than the average because of the skew in channel sizes in the Lightning topology: most transactions use large channels while a few unfortunate large transactions need more time to reuse tokens from smaller channels. Yet, the largest Spider transactions experience at most 2 seconds of additional delay when

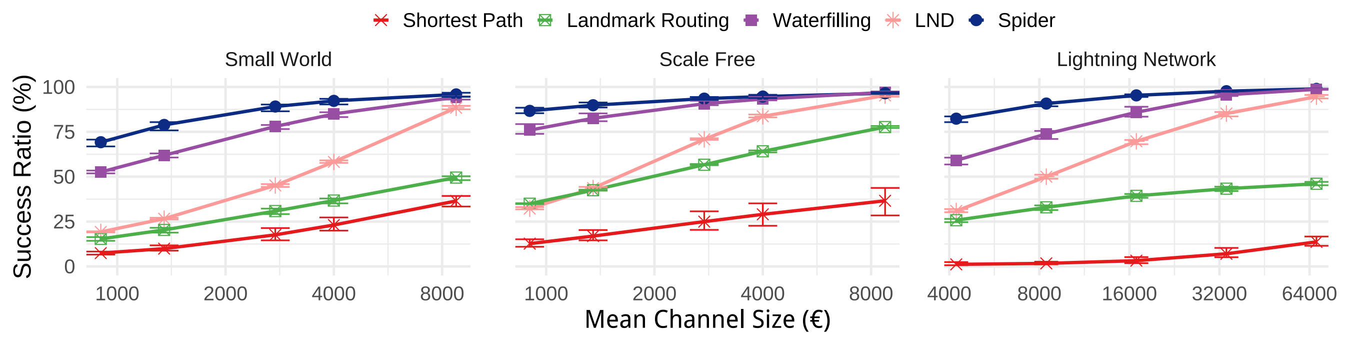


Figure 9: Performance of different algorithms on small-world, scale-free and Lightning Network topologies, for different per sender transaction arrival rates. Spider consistently outperforms all other schemes achieving near 100% average success ratio. Note the log scale of the x-axes.

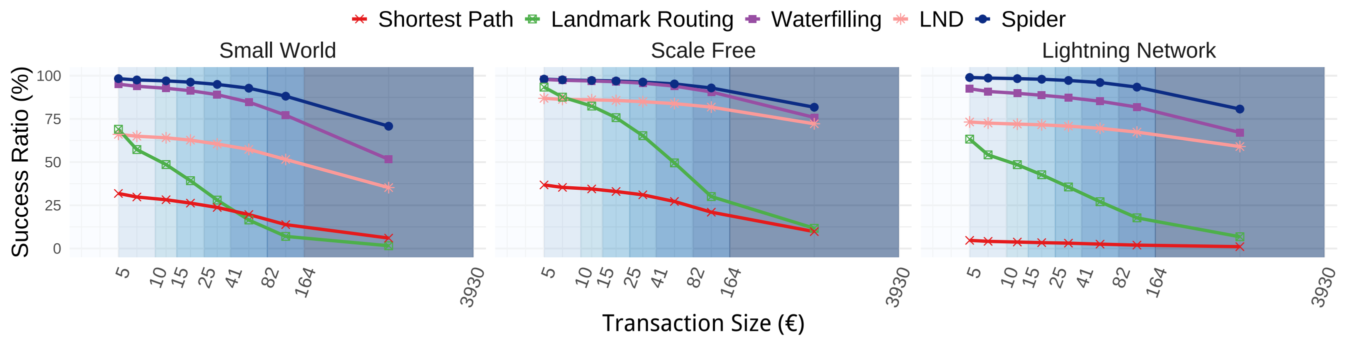


Figure 10: Breakdown of performance of different schemes by size of transactions completed. Each point reports the success ratio for transactions whose size belongs to the interval denoted by the shaded region. Each interval corresponds roughly to a 12.5% weight in the transaction size CDF shown in Fig. 7a. The graphs correspond to the midpoints of the corresponding Lightning sampled channel sizes in Fig. 9.

compared to LND, a small hit relative to the 20% increase in overall success ratio at a mean channel size of 16880€. LND's latency also increases with size since it retries transactions, often upto 10 times until it finds a single path with enough capacity. In contrast, Landmark Routing and Shortest path are size-agnostic in their path-choice for transactions.

Waterfilling pauses transactions when there is no available balance and resumes sending when balance becomes available. Small transactions are unlikely to be paused in their lifetime while mid-size transactions are paused a few times before they complete. In contrast, large transactions are likely to be paused many times, eventually getting canceled if paused too much. This has two implications: (i) the few large transactions that are successful with Waterfilling are not paused much and contribute smaller latencies than mid-size transactions, and (ii) Waterfilling's conservative pause and send mechanism implies there is less contention for the large transactions that are actually sent into the network, leading to smaller latencies than what they experience with Spider.

7.4 Effect of DAGs

Real transaction demands are often not pure circulations: consumer nodes spend more, and merchant nodes receive

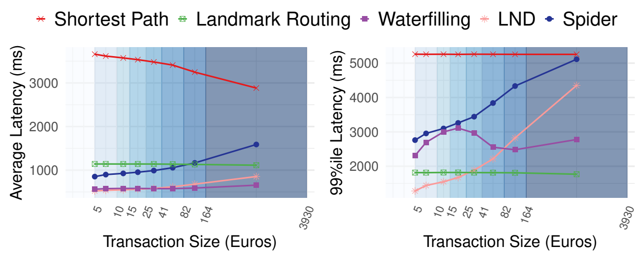


Figure 11: Average and 99%ile transaction latency for different routing schemes on the Lightning topology. Transactions experience 1-2s of additional latency with Spider relative to LND for a 20% improvement in throughput.

more. To simulate this, we add 5 DAG payment graphs (§7.1) to circulation payment graphs, varying the relative weight to generate effectively 5%, 20% and 40% DAG in the total demand matrix. We run all schemes on the Lightning topology with a mean channel size of 16880€; results on the synthetic topologies are in App. C.4.

Fig. 12 shows the success ratio and normalized throughput. We immediately notice that no scheme achieves the theoretical upper bound on throughput (i.e., the % circulation demand). However, throughput is closer to the bound when there is a smaller DAG component in the demand matrix. This suggests

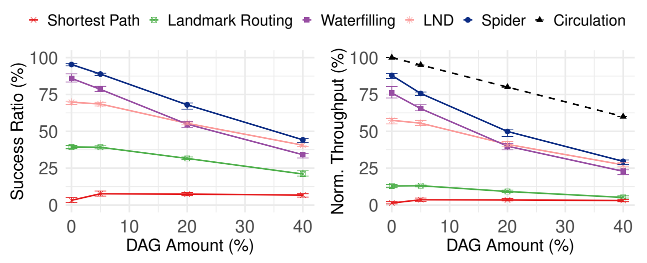


Figure 12: Performance of different algorithms on the Lightning topology as the DAG component in the transaction demand matrix is varied. As the DAG amount is increased, the normalized throughput achieved is further away from the expected optimal circulation throughput.

that not only is the DAG itself unroutable, it also alters the PCN balances in a way that prevents the circulation from being fully routed. Further, the more DAG there is, the more affected the circulation is. This is because the DAG causes a deadlock (§3).

To illustrate this, we run two scenarios: (i) a pure circulation demand X for 3000s, and (ii) a traffic demand $(X+Y)$ containing 20% DAG for 2000s followed by the circulation X for 1000s after that. Here, each sender sends 200€/s of unit-sized transactions in X. We observe a time series of the normalized throughput over the 3000s. The mean channel size is 4000€ and 16990€ for the synthetic and real topologies respectively.

Fig. 13 shows that Spider achieves 100% throughput (normalized by the circulation demand) at steady state for the pure circulation demand on all topologies. However, when the DAG component is introduced to the demand, it affects the topologies differently. Firstly, we do not observe the expected 80% throughput for the circulation in the presence of the DAG workload suggesting that the DAG affects the circulation. Further, even once the circulation demand is restored for the last 1000s, in the scale free and Lightning Network topology, the throughput achieved is no longer 100%. In other words, in these two topologies, the DAG causes a deadlock that affects the circulation even after the DAG is removed.

As described in §3, the solution to this problem involves replenishing funds via on-chain rebalancing, since DAG demands continuously move money from sources to sinks. We therefore implement a simple rebalancing scheme where every router periodically reallocates funds between its payment channels to equalize their *available balance*. The frequency of rebalancing for a router, is defined by the number of successful transaction-units (in €) between consecutive rebalancing events. In this model, the frequency captures the on-chain rebalancing cost vs. routing fee trade-off for the router.

Fig. 14 shows the success ratio and normalized throughput achieved by different schemes when rebalancing is enabled for the traffic demand with 20% DAG from Fig. 12, or Fig. 13. Spider is able to achieve 90% success ratio even when its routers rebalance only every 10,000€ routed while LND is never able to sustain more than 85% success ratio even when rebalancing for every 10€ routed. This is because LND deems

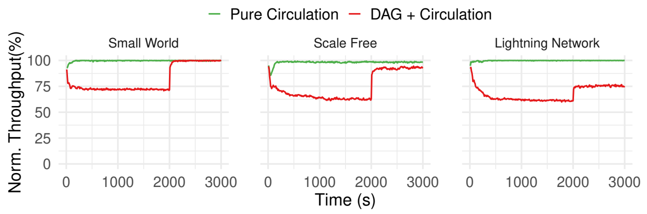


Figure 13: Comparing throughput when a pure circulation demand is run for 3000s to a scenario where a circulation demand is restored for 1000s after 2000s of a demand with 20% DAG. The throughput achieved on the last 1000s of circulation is not always the expected 100% even after the DAG is removed.

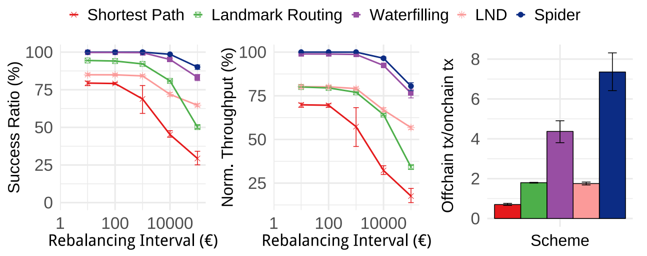


Figure 14: Performance of different algorithms on the Lightning topology when augmented with on-chain rebalancing. Spider needs less frequent rebalancing to sustain high throughput. Spider offloads 3-4x more transactions onto a PCN per blockchain transaction than LND.

a channel unusable for 5 seconds every time a transaction fails on it due to lack of funds and this is further worsened by its lack of transaction splitting. This implies that when using Spider, routers need to pay for only one on-chain transaction typically costing under 1€ [7] for every 10,000€ routed. Thus, for a router to break even, it would have to charge 1€ for every 10000€ routed. This translates into significantly lower routing fees for end-users than today's payment systems [12]. Fig. 14 also captures the same result in the form of the best offloading or number of off-chain PCN transactions per blockchain transaction achieved by each algorithm. Transactions that fail on the PCN as well as rebalancing transactions are counted towards the transactions on the blockchain. Spider is able to route 7-8 times as many transactions off-chain for every blockchain transaction, a 4x improvement from the state-of-the-art LND.

7.5 Spider's Design Choices

In this section, we investigate Spider's design choices with respect to the number of paths, type of paths, and the scheduling algorithm that services transaction-units at Spider's queues. We evaluate these on both the real and synthetic topologies with channel sizes sampled from the LCSD, and scaled to have mean of 16880€ and 4000 € respectively .

Choice of Paths. We vary the type of paths that Spider uses by replacing edge-disjoint widest paths with edge-disjoint shortest paths, Yen's shortest paths [60], oblivious paths [48] and

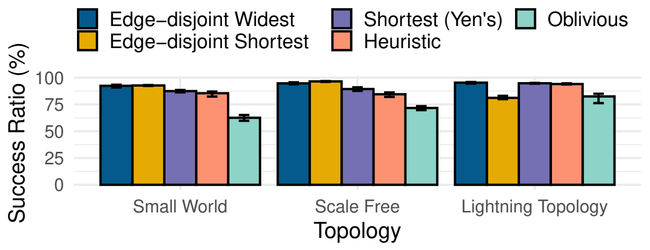


Figure 15: Performance of Spider as the type of paths considered per sender-receiver pair is varied. Edge-disjoint widest outperforms others by 1-10% on the Lightning Topology without being much worse on the synthetic topologies.

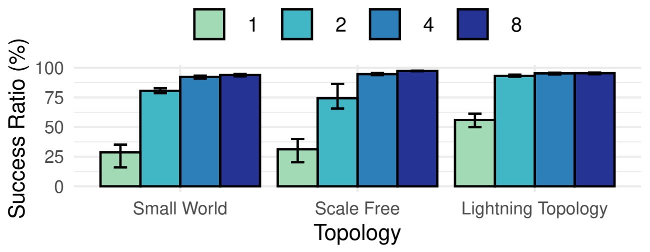


Figure 16: Performance of Spider as the number of edge-disjoint widest paths considered per sender-receiver pair is varied on different topologies. Increasing the number of paths increases success ratio, but the gains are low in going from 4 to 8 paths.

a heuristic approach. For the widest and oblivious path computations, the channel size acts as the edge weight. The heuristic picks 4 paths for each flow with the highest bottleneck balance/RTT value. Fig. 15 shows that edge-disjoint widest paths outperforms other approaches by 1-10% on the Lightning Topology while being only 1-2% worse that edge-disjoint shortest paths on the synthetic topologies. This is because widest paths are able to utilize the capacity of the network better when there is a large skew (Fig. 7b) in payment channel sizes.

Number of Paths. We vary the maximum number of edge-disjoint widest paths Spider allows from 1 to 8. Fig. 16 shows that, as expected, the success ratio increases with an increase in number of paths, as more paths allow Spider to better utilize the capacity of the PCN. While moving from 1 to 2 paths results in 30-50% improvement in success ratio, moving from 4 to 8 paths has negligible benefits (<5%). This is because the sparseness of the three PCN topologies causes most flows to have at most 5-6 edge-disjoint widest paths. Further, Spider prefers paths with smaller RTTs since they receive feedback faster resulting in the shortest paths contributing most to the overall rate for the flow. As a result, we use 4 paths for Spider.

Scheduling Algorithms. We modify the scheduling algorithm at the per-destination queues at the sender as well as the router queues in Spider to process transactions as per First-In-First-Out (FIFO), Earliest-Deadline-First (EDF) and Smallest-Payment-First (SPF) in addition to the LIFO baseline. Fig. 17 shows that LIFO achieves a success ratio that is 10-28% higher than its counterparts. This is because

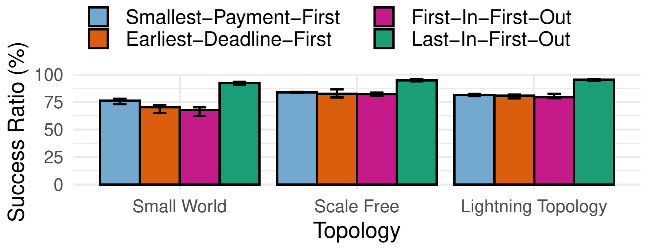


Figure 17: Performance of Spider as the scheduling algorithm at the sender and router queues is varied. Last in first out outperforms all other approaches by over 10% on all topologies.

LIFO prioritizes transactions that are newest or furthest from their deadlines and thus, most likely complete especially when the PCNs is overloaded. Spider's rate control results in long wait times in the sender queues themselves. This causes FIFO and EDF that send out transactions closest to their deadlines to time out immediately in the network resulting in poor throughput. When SPF deprioritizes large payments at router queues, they consume funds from other payment channels for longer, reducing the effective capacity of the network.

7.6 Additional Results

In addition to the results described so far, we run additional experiments that are described in the Appendices.

1. We compare Spider to Celer, as proposed in a white-paper [11], and show that Spider outperforms Celer's success ratio by 2x on a scale free topology with 10 nodes and 25 edges (App. C.1).
2. We evaluate the schemes on the synthetic and real topologies with a simpler channel size distribution where all channels have equal numbers of tokens. Even in this scenario, Spider is able to successfully route more than 95% of the transactions with less than 25% of the capacity required by LND (App. C.2).
3. We evaluate the schemes for their fairness across multiple payments and show that Spider does not hurt small payments to gain on throughput (App. C.3).
4. We show the effect of DAG workloads on synthetic topologies. In particular, we identify deadlocks with those topologies too and show that Spider requires rebalancing only every 10,000€ successfully routed to sustain high success ratio and normalized throughput (App. C.4).

8 Related Work

PCN Improvements. Nodes in current Lightning Network implementations, maintain a local view of the network topology and source-route transactions along the shortest path [2, 15]. Classical max-flow-based alternatives are impractical for the Lightning Network that has over 5000 nodes and 30,000 channels [9, 16] due to their computational complexity. Recent proposals have used a modified version of max-flow that differentiates based on the size of transactions [57]. However, inferring the size of payments is hard in an onion-routed network like Lightning.

Two main alternatives to max-flow routing have been proposed: landmark routing and embedding-based routing. In *landmark routing*, select routers (landmarks) store routing tables for the rest of the network, and nodes only route transactions to a landmark [55]. This approach is used in Flare [47] and SilentWhispers [42, 44]. *Embedding-based* or *distance-based* routing learns a vector embedding for each node, such that nodes that are close in network hop distance are also close in embedded space. Each node relays each transaction to the neighbor whose embedding is closest to the destination's embedding. VOUTE [49] and SpeedyMurmurs [50] use embedding-based routing. Computing and updating the embedding dynamically as the topology and link balances change is a primary challenge of these approaches. Our experiments and prior work [51] show that Spider outperforms both approaches.

PCN improvements outside of the routing layer focus on rebalancing existing payment channels more easily [28, 39]. Revive [39] leverages cycles within channels wanting to rebalance and initiates balancing off-chain payments between them. These techniques are complementary to Spider and can be used to enhance overall performance. However, §7.4 shows that a more general rebalancing scheme that moves funds at each router independently fails to achieve high throughput without a balanced routing scheme.

Utility Maximization and Congestion Control. Network Utility Maximization (NUM) is a popular framework for developing decentralized transport protocols in data networks to optimize a fairness objective [37]. NUM uses link "prices" derived from the solution to the utility maximization problem, and senders compute rates based on these router prices. Congestion control algorithms that use buffer sizes or queuing delays as router signals [22,30,53] are closely related. While the Internet congestion control literature has focused on links with fairly stable capacities, this paper shows that they can be effective even in networks with capacities dependent on the input rates themselves. Such problems have also been explored in the context of ride-sharing, for instance [24, 25], and require new innovation in both formulating and solving routing problems.

9 Conclusion

We motivate the need for efficient routing on PCNs and propose Spider, a protocol for balanced, high-throughput routing in PCNs. Spider uses a packet-switched architecture, multi-path congestion control, and and in-network scheduling. Spider achieves nearly 100% throughput on circulation payment demands across both synthetic and real topologies. We show how the presence of DAG payments causes deadlocks that degrades circulation throughput, necessitating on-chain intervention. In such scenarios, Spider is able to support 4x more transactions than the state-of-the-art on the PCN itself.

This work shows that Spider needs less on-chain rebalancing to relieve deadlocked PCNs. However, it remains to be seen if deadlocks can be prevented altogether. Spider relies on routers signaling queue buildup correctly to the senders, but this work does not analyze incentive compatibility for rogue routers aiming to maximize fees. A more rigorous treatment of the privacy implications of Spider routers relaying queuing delay is left to future work.

Acknowledgments

We thank Andrew Miller, Thaddeus Dryja, Evan Schwartz, Vikram Nathan, and Aditya Akella for their detailed feedback. We also thank the Sponsors of Fintech@CSAIL, the Initiative for CryptoCurrencies and Contracts (IC3), Distributed Technologies Research Foundation, the Cisco Research Center, the National Science Foundation grants CNS-1718270, CNS-1563826, CNS-1910676, CCF-1705007 and CNS-1617702, and the Army Research Office under grant W911NF1810332 for their support.

References

[1] http://omnetpp.org/.

[2] Amount-independent payment routing in Lightning Networks. https://medium.com/coinmonks/amount-independent-payment-routing-in-lightning-networks-6409201ff5ed.

[3] AMP: Atomic Multi-Path Payments over Lightning. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-February/000993.html.

[4] Barabasi Albert Graph. https://networkx.github.io/documentation/networkx-1.9.1/reference/generated/networkx.generators.random_graphs.barabasi_albert_graph.html.

[5] Bitcoin Core. https://bitcoin.org/en/bitcoin-core/.

[6] Bitcoin Core Daemon. https://bitcoin.org/en/full-node#other-linux-daemon.

[7] Bitcoin historical fee chart. https://bitinfocharts.com/comparison/bitcoin-median_transaction_fee.html.

[8] Bitcoin Regtest Mode. https://bitcoin.org/en/developer-examples#regtest-mode.

[9] Blockchain caffe. https://blockchaincaffe.org/map/.

[10] c-lightning: A specification compliant Lightning Network implementation in C. https://github.com/ElementsProject/lightning.

[11] Celer Network: Bring Internet Scale to Every Blockchain. https://www.celer.network/doc/CelerNetwork-Whitepaper.pdf.

[12] Credit Card Merchant Processing Fees. https://paymentdepot.com/blog/average-credit-card-processing-fees/.

[13] etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://github.com/etcd-io/etcd.

[14] Ethereum. https://www.ethereum.org/.

[15] Lightning Network Daemon. https://github.com/lightningnetwork/lnd.

[16] Lightning Network Search and Analysis Engine. https://1ml.com.

[17] Onion Routed Micropayments for the Lightning Network. https://github.com/lightningnetwork/lightning-onion.

[18] Raiden network. https://raiden.network/.

[19] The NewReno Modification to TCP's Fast Recovery Algorithm. https://tools.ietf.org/html/rfc6582.

[20] Watts Strogatz Graph. https://networkx.github.io/documentation/networkx-1.9/reference/generated/networkx.generators.random_graphs.watts_strogatz_graph.html.

[21] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993.

[22] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). *ACM SIGCOMM computer communication review*, 41(4):63–74, 2011.

[23] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath. Deconstructing the blockchain to approach physical limits. *arXiv preprint arXiv:1810.08092*, 2018.

[24] S. Banerjee, R. Johari, and C. Riquelme. Pricing in ride-sharing platforms: A queueing-theoretic approach. In *Proceedings of the Sixteenth ACM Conference on Economics and Computation*, pages 639–639. ACM, 2015.

[25] S. Banerjee, R. Johari, and C. Riquelme. Dynamic pricing in ridesharing platforms. *ACM SIGecom Exchanges*, 15(1):65–70, 2016.

[26] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[27] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[28] C. Burchert, C. Decker, and R. Wattenhofer. Scalable funding of bitcoin micropayment channel networks. *Royal Society open science*, 5(8):180089, 2018.

[29] C. N. Cordi. *Simulating high-throughput cryptocurrency payment channel networks*. PhD thesis, 2017.

[30] N. Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Citeseer, 2008.

[31] A. Eryilmaz and R. Srikant. Joint congestion control, routing, and mac for stability and fairness in wireless networks. *IEEE Journal on Selected Areas in Communications*, 24(8):1514–1524, 2006.

[32] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[33] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[34] U. M. L. Group. Credit card fraud detection, 2018. https://www.kaggle.com/mlg-ulb/creditcardfraud.

[35] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[36] P. Hu and W. C. Lau. A survey and taxonomy of graph sampling. *arXiv preprint arXiv:1308.5865*, 2013.

[37] F. Kelly and T. Voice. Stability of end-to-end algorithms for joint routing and rate control. *ACM SIGCOMM Computer Communication Review*, 35(2):5–12, 2005.

[38] F. P. Kelly, A. K. Maulloo, and D. K. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998.

[39] R. Khalil and A. Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 439–453. ACM, 2017.

[40] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[41] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. Wolff. A brief history of the internet. *SIGCOMM Comput. Commun. Rev.*, 39(5):22–31, Oct. 2009.

[42] G. Malavolta, P. Moreno-Sanchez, A. Kate, and M. Maffei. SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks. *IACR Cryptology ePrint Archive*, 2016:1054, 2016.

[43] R. McManus. Blockchain speeds & the scalability debate. *Blocksplain*, February 2018.

[44] P. Moreno-Sanchez, A. Kate, M. Maffei, and K. Pecina. Privacy preserving payments in credit networks. In *Network and Distributed Security Symposium*, 2015.

[45] D. P. Palomar and M. Chiang. A tutorial on decomposition methods for network utility maximization. *IEEE Journal on Selected Areas in Communications*, 24(8):1439–1451, 2006.

[46] J. Poon and T. Dryja. The Bitcoin Lightning Network: Scalable Off-chain Instant Payments. *draft version 0.5*, 9:14, 2016.

[47] P. Prihodko, S. Zhigulin, M. Sahno, A. Ostrovskiy, and O. Osuntokun. Flare: An approach to routing in lightning network. 2016.

[48] H. Racke. Minimizing congestion in general networks. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 43–52. IEEE, 2002.

[49] S. Roos, M. Beck, and T. Strufe. Anonymous addresses for efficient and resilient routing in f2f overlays. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[50] S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg. Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions. *arXiv preprint arXiv:1709.05748*, 2017.

[51] V. Sivaraman, S. B. Venkatakrishnan, M. Alizadeh, G. Fanti, and P. Viswanath. Routing cryptocurrency with the spider network. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 29–35. ACM, 2018.

[52] R. Srikant. *The mathematics of Internet congestion control*. Springer Science & Business Media, 2012.

[53] C.-H. Tai, J. Zhu, and N. Dukkipati. Making large scale deployment of rcp practical for real networks. In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pages 2180–2188. IEEE, 2008.

[54] S. Thomas and E. Schwartz. A protocol for interledger payments. *URL https://interledger. org/interledger. pdf*, 2015.

[55] P. F. Tsuchiya. The landmark hierarchy: a new hierarchy for routing in very large networks. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 35–42. ACM, 1988.

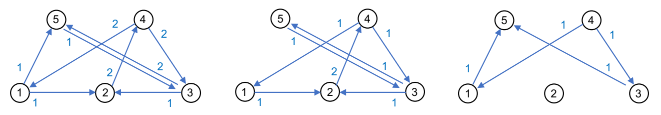
[56] Visa. Visa acceptance for retailers. `https://usa.visa.com/run-your-business/small-business-tools/retail.html`.

[57] P. Wang, H. Xu, X. Jin, and T. Wang. Flash: efficient dynamic routing for offchain networks. *arXiv preprint arXiv:1902.05260*, 2019.

[58] D. Wischik, M. Handley, and M. B. Braun. The resource pooling principle. *ACM SIGCOMM Computer Communication Review*, 38(5):47–52, 2008.

[59] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, volume 11, pages 8–8, 2011.

[60] J. Y. Yen. Finding the k shortest loopless paths in a network. *management Science*, 17(11):712–716, 1971.



(a) Payment graph (b) Circulation (c) DAG

Figure 18: Example payment graph (denoted by blue lines) for a five node network (left). It decomposes into a maximum circulation and DAG components as shown in (b) and (c).

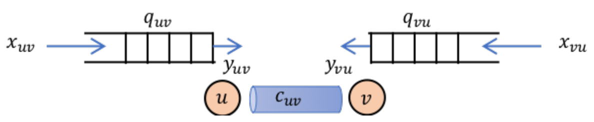


Figure 19: Model of queues at a payment channel between nodes u and v. x_{uv} and y_{uv} denote the rates at which transaction-units for v arrive into and get serviced at the queue at u respectively. c_{uv} is the capacity of the payment channel and q_{uv} denotes the total number of transaction-units waiting in u's queue to be serviced.

Appendices

A Circulations and Throughput Bounds

For a network $G(V,E)$ with set of routers V, we define a *payment graph* $H(V,E_H)$ as a graph that specifies the payment demands between different users. The weight of any edge (i,j) in the payment graph is the average rate at which user i seeks to transfer funds to user j. A *circulation graph* $C(V,E_C)$ of a payment graph is any subgraph of the payment graph in which the weight of an edge (i,j) is at most the weight of (i,j) in the payment graph, and moreover the total weight of incoming edges is equal to the total weight of outgoing edges for each node. Of particular interest are *maximum circulation graphs* which are circulation graphs that have the highest total demand (i.e., sum of edge weights), among all possible circulation graphs. A maximum circulation graph is not necessarily unique for a given payment graph.

Proposition 1. *Consider a payment graph H with a maximum circulation graph C^*. Let $\nu(C^*)$ denote the total demand in C^*. Then, on a network in which each payment channel has at least $\nu(C^*)$ units of escrowed funds, there exists a balanced routing scheme that can achieve a total throughput of $\nu(C^*)$. However, no balanced routing scheme can achieve a throughput greater than $\nu(C^*)$ on any network.*

Proof. Let $w_{C^*}(i,j)$ denote the payment demand from any user i to user j in the maximum circulation graph C^*. To see that a throughput of $\nu(C^*)$ is achievable, consider routing the circulation demand along the shortest paths of any spanning tree T of the payment network G. In this routing, for any pair of nodes $i,j \in V$ there exists a unique path from i to j in T through which $w_{C^*}(i,j)$ amount of flow is routed. We claim that such a routing scheme is perfectly balanced on all the links. This is because for any partition $S,V\backslash S$ of C^*, the net flow going from S to $V\backslash S$ is equal to the net flow going from $V\backslash S$ to S in C^*. Since the flows along an edge e of T correspond precisely to the net flows across the partitions obtained by removing e in T , it follows that the flows on e are balanced as well. Also, for any flow (i,j) in the demand graph C^*, the shortest path route from i to j in T can cross an edge e at most once. Therefore the total amount of flow going through an edge is at most the total amount of flow in C^*, which is $\nu(C^*)$.

Next, to see that no balanced routing scheme can achieve a throughput greater than $\nu(C^*)$, assume the contrary and suppose there exists a balanced routing scheme SCH with a throughput greater than $\nu(C^*)$. Let $H_{\texttt{SCH}} \subseteq H$ be a payment graph where the edges represent the portion of demand that is actually routed in SCH. Since $\nu(H_{\texttt{SCH}}) > \nu(C^*)$, $H_{\texttt{SCH}}$ is not a circulation and there exists a partition $S,V\backslash S$ such that the net flow from S to $V\backslash S$ is strictly greater than the net flow from $V\backslash S$ to S in $H_{\texttt{SCH}}$. However, the net flows routed by SCH across the same partition $S,V\backslash S$ in G are balanced (by assumption) resulting in a contradiction. Thus we conclude there does not exist any balanced routing scheme that can achieve a throughput greater than $\nu(C^*)$. □

B Optimality of Spider

B.1 Fluid Model

In this section we describe a fluid model approximation of the system dynamics under Spider's protocol. Following a similar notation as in §5, for a path p we let $x_p(t)$ denote the rate of flow on it at time t. For a channel (u,v) and time t, let $q_{u,v}(t)$ be the size of the queue at router u, $f_{u,v}(t)$ be the fraction of incoming packets that are marked at u, $x_{u,v}(t)$ be the total rate of incoming flow at u, and $y_{u,v}(t)$ be the rate at which transactions are serviced (*i.e.,* forwarded to router v) at u. All variables are real-valued. We approximate Spider's dynamics via the following system of equations

$$\dot{x}_p(t) = \left[\frac{x_p(t)}{\sum_{p' \in \mathcal{P}_{i_p,j_p}} x_{p'}(t)} - \sum_{(u,v)\in p} f_{u,v}(t) x_p(t) \right]^+_{x_p(t)} \quad \forall p \in \mathcal{P} \tag{8}$$

$$\dot{q}_{u,v}(t) = [x_{u,v}(t) - y_{u,v}(t)]^+_{q_{u,v}(t)} \quad \forall (u,v) \in E \tag{9}$$

$$\dot{f}_{u,v}(t) = [q_{u,v}(t) - q_{\text{thresh}}]^+_{f_{u,v}(t)} \quad \forall (u,v) \in E, \tag{10}$$

where $y_{u,v}(t) = y_{v,u}(t) =$

$$\begin{cases} \frac{c_{u,v}}{2\Delta} & \text{if } q_{u,v}(t) > 0 \;\&\; q_{v,u}(t) > 0 \\ \min\{\frac{c_{u,v}}{2\Delta}, x_{v,u}(t)\} & \text{if } q_{u,v}(t) > 0 \;\&\; q_{v,u}(t) = 0 \\ \min\{\frac{c_{u,v}}{2\Delta}, x_{u,v}(t)\} & \text{if } q_{u,v}(t) = 0 \;\&\; q_{v,u}(t) > 0 \\ \min\{\frac{c_{u,v}}{2\Delta}, x_{u,v}(t), x_{v,u}(t)\} & \text{if } q_{u,v}(t) = 0 \;\&\; q_{v,u}(t) = 0 \end{cases} \tag{11}$$

for each $(u,v) \in E$. Let i_p and j_p denote the source and destination nodes for path p respectively. Then, $\mathcal{P}_{i_p,j_p}$ denotes the set of all paths i_p uses to route to j_p. Equation (8) models how the rate on a path p increases upon receiving successful acknowledgements or decreases if the packets are marked, per Equations (6) and (7) in §6.3. If the fraction of packets marked at each router is small, then the aggregate fraction of packets that return marked on a path p can be approximated by the sum $\sum_{(u,v)\in p} f_{u,v}$ [52]. Hence the rate which marked packets arrive for a path p is $\sum_{(u,v)\in p} f_{u,v} x_p$. Similarly, the rate which successful acknowledgements are received on a path p is $x_p(1 - \sum_{(u,v)\in p} f_{u,v})$, which can be approximated as simply x_p if the marking fractions are small. Since Spider increases the window by $1/(\sum_{p'\in\mathcal{P}_{i_p,j_p}} w_{p'})$ for each successful acknowledgement received, the average rate at which x_p increases is $x_p/(\sum_{p'\in\mathcal{P}_{i_p,j_p}} x_{p'})$. Lastly, the rate x_p cannot become negative; so if $x_p = 0$ we disallow $\dot{x}_p$ from being negative. The notation $(x)_y^+$ means x if $y > 0$ and 0 if $y = 0$.

Equations (9) and (10) model how the queue sizes and fraction of packets marked, respectively, evolve at the routers. For a router u in payment channel (u,v), by definition $y_{u,v}$ is the rate at which transactions are serviced from the queue $q_{u,v}$, while transactions arrive at the queue at a rate of $x_{u,v}$ (Figure 19). Hence the net rate at which $q_{u,v}$ grows is given by the difference $x_{u,v} - y_{u,v}$. The fraction of packets marked at a queue grows if the queue size is larger than a threshold q_{thresh}, and drops otherwise, as in Equation (10). This approximates the marking model of Spider (§6.2) in which packets are marked at a router if their queuing delay exceeds a threshold.

To understand how the service rate $y_{u,v}$ evolves (Equation (11)), we first make the approximation that the rate at which transactions are serviced from the queue at a router u is equal to the rate at which tokens are replenished at the router, *i.e.*, $y_{u,v} = y_{v,u}$ for all $(u,v) \in E$. The precise value for $y_{u,v}$ at any time, depends on both the arrival rates and current occupancy of the queues at routers u and v. If both $q_{u,v}$ and $q_{v,u}$ are non-empty, then there are no surplus of tokens available within the channel. A token when forwarded by a router is unavailable for Δ time units, until its acknowledgement is received. Therefore the maximum rate at which tokens on the channel can be forwarded is $c_{u,v}/\Delta$, implying $y_{u,v} + y_{v,u} = c_{u,v}$ or $y_{u,v} = y_{v,u} = c_{u,v}/(2\Delta)$ in this case. If $q_{u,v}$ is non-empty and $q_{v,u}$ is empty, then there are no surplus tokens available at u's end. Router v however may have tokens available, and service transactions at the same rate at which they are arriving, *i.e.*, $y_{v,u} = x_{v,u}$. This implies tokens become available at router u at a rate of $x_{v,u}$ and hence $y_{u,v} = x_{v,u}$. However, if the transaction arrival rate $x_{v,u}$ is too large at v, it cannot service them at a rate more than $c_{u,v}/(2\Delta)$ and a queue would start building up at $q_{v,u}$. The case where $q_{u,v}$ is empty and $q_{v,u}$ is non-empty follows by interchanging the variables u and v in the description above. Lastly, if both $q_{u,v}$ and $q_{v,u}$ are empty, then the service rate $y_{u,v}$ can at most be equal to the arrival rate $x_{v,u}$. Similarly $y_{v,u}$ can be at most $x_{u,v}$. Since $y_{u,v} = y_{v,u}$ by our approximation, we get the expression in Equation (11).

We have not explicitly modeled delays, and have made simplifying approximations in the fluid model above. Nevertheless this model is useful for gaining intuition about the first-order behavior of the Spider protocol. In the following section, we use this model to show that Spider finds optimal rate allocations for a parallel network topology.

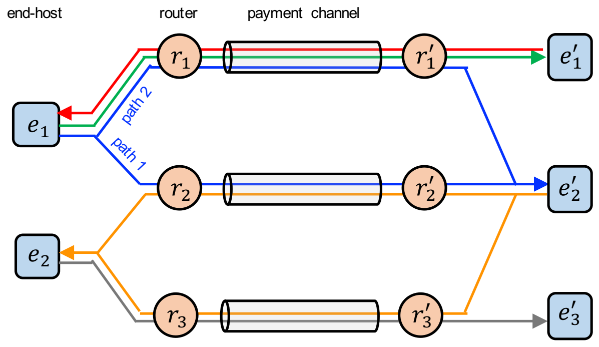


Figure 20: Example of a parallel network topology with bidirectional flows on each payment channel.

B.2 Proof of Optimality

Consider a PCN comprising of two sets of end-hosts $\{e_1,\ldots,e_m\}$ and $\{e'_1,\ldots,e'_n\}$ that are connected via k parallel payment channels $(r_1,r'_1),\ldots,(r_k,r'_k)$ as shown in Figure 20. The end-hosts from each set have demands to end-hosts on the other set. The end-hosts within a set, however, do not have any demands between them. Let the paths for different source-destination pairs be such that for each path p, if p contains a directed edge (r_i, r'_i) for some i then there exists another path (for a different source-destination pair) that contains the edge (r'_i, r_i). We will show that running Spider on this network results in rate allocations that are an optimal solution to the optimization problem in Equations (1)–(5). Under a fluid model for Spider as discussed in §B.1, assuming convergence, we observe that in the steady-state the time derivatives of the rate of flow of each path (Equation (8)) must be non-positive, *i.e.*,

$$\frac{1}{\sum_{p'\in\mathcal{P}_{i_p,j_p}} x^*_{p'}} - \sum_{(u,v)\in p} f^*_{u,v} \begin{cases} = 0 & \text{if } x^*_p > 0 \\ \le 0 & \text{if } x^*_p = 0 \end{cases} \quad \forall p \in \mathcal{P}, \quad (12)$$

where the superscript * denotes values at convergence (*e.g.*, x^*_p is the rate of flow on path p at convergence). Similarly, the rate of growth of the queues must be non-positive, or

$$x^*_{u,v} \begin{cases} = y^*_{u,v} & \text{if } q^*_{u,v} > 0 \\ \le y^*_{u,v} & \text{if } q^*_{u,v} = 0 \end{cases} \quad \forall (u,v) \in E. \quad (13)$$

Now, consider the optimization problem in Equations (1)–(5) for this parallel network. For simplicity we will assume the sender-receiver demands are not constrained. From Equation (13) above, the transaction arrival rates $x^*_{u,v}$ and $x^*_{v,u}$ for a channel (u,v) satisfy the capacity constraints in Equation (3). This is because $x^*_{u,v} \le y^*_{u,v}$ from Equation (13) and $y_{u,v}(t)$ is at most $\frac{c_{u,v}}{2\Delta}$ from Equation (11). Similarly the transaction arrival rates also satisfy the balance constraints in Equation (4). To see this, we first note the that the queues on all payment channels through which a path (corresponding to a sender-receiver pair) passes must be non-empty. For otherwise, if a queue $q^*_{u,v}$ is empty then the fraction of marked packets on a path p through (u,v) goes to 0, and the rate of flow x^*_p would increase as per Equation (8). Therefore we have $x^*_{u,v} = y^*_{u,v}$ (from Equation (13)) for every channel. Combining this with $y_{u,v}(t) = y_{v,u}(t)$ (Equation (11)), we conclude that the arrival rates are balanced on all channels. Thus the equilibrium rates $\{x^*_p : p \in \mathcal{P}\}$ resulting from Spider are in the feasible set for the routing optimization problem.

Next, let $\lambda_{u,v} \geq 0$ and $\mu_{u,v} \in \mathbb{R}$ be the dual variables corresponding to the capacity and balance constraints, respectively, for a channel (u, v). Consider the following mapping from $f^*_{u,v}$ to $\lambda_{u,v}$ and $\mu_{u,v}$

$$\lambda^*_{u,v} \leftarrow (f^*_{u,v} + f^*_{v,u})/2 \quad \forall (u,v) \in E \tag{14}$$

$$\mu^*_{u,v} \leftarrow f^*_{u,v}/2 \quad \forall (u,v) \in E, \tag{15}$$

where the superscript * on the dual variables indicate that they have been derived from the equilibrium states of the Spider protocol. Since $f_{u,v}(t)$ is always non-negative (Equation (10)), we see that $\lambda^*_{u,v} \geq 0$ for all (u,v). Therefore $\{\lambda^*_{u,v} : (u,v) \in E\}$ and $\{\mu^*_{u,v} : (u,v) \in E\}$ are in the feasible set of the dual of the routing optimization problem.

Next, we have argued previously that the queues on all payment channels through which a path (corresponding to a sender-received pair) passes must be non-empty. While we used this observation to show that the channel rates $x^*_{u,v}$ are balanced, it also implies that the rates are at capacity, *i.e.*, $x^*_{u,v} = c_{u,v}/(2\Delta)$, or $x^*_{u,v} + x^*_{v,u} = c_{u,v}/\Delta$ for all (u,v). This directly follows from Equation (13) and the first sub-case in Equation (11). It follows that the primal variables $\{x^*_p : p \in \mathcal{P}\}$ and the dual variables $\{\lambda^*_{u,v} : (u,v) \in E\}, \{\mu^*_{u,v} : (u,v) \in E\}$ satisfy the complementary slackness conditions of the optimization problem.

Last, the optimality condition for the primal variables on the Lagrangian defined with dual variables $\{\lambda^*_{u,v} : (u,v) \in E\}$ and $\{\mu^*_{u,v} : (u,v) \in E\}$ stipulates that

$$\frac{1}{\sum_{p' \in \mathcal{P}_{i_p,j_p}} x_{p'}} - \sum_{(u,v) \in p} (\lambda^*_{u,v} + \mu^*_{u,v} - \mu^*_{v,u}) \begin{cases} = 0 & \text{if } x_p > 0 \\ \leq 0 & \text{if } x_p = 0 \end{cases}, \tag{16}$$

for all $p \in \mathcal{P}$. However, note that for any path p

$$\begin{aligned}\sum_{(u,v) \in p} (\lambda^*_{u,v} + \mu^*_{u,v} - \mu^*_{v,u}) &= \sum_{(u,v) \in p} \frac{f^*_{u,v} + f^*_{v,u}}{2} + \frac{f^*_{u,v}}{2} - \frac{f^*_{v,u}}{2} \\ &= \sum_{(u,v) \in p} f^*_{u,v},\end{aligned} \tag{17}$$

where the first equation above follows from our mapping for $\lambda^*_{u,v}$ and $\mu^*_{u,v}$ in Equations (14), (15). Combining this with Equation (12), we see that $x_p \leftarrow x^*_p$ for all $p \in \mathcal{P}$ is

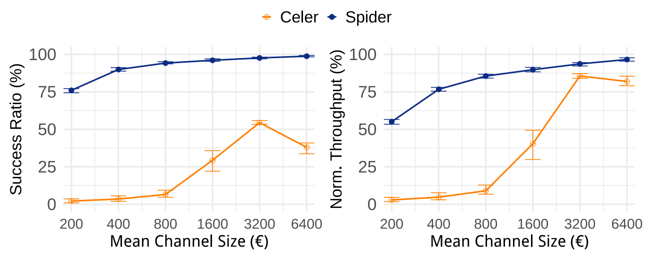


Figure 21: Spider's performance relative to Celer on a 10 node scale free topology. Spider achieves a 2x improvement in success ratio even at Celer's peak performance. Celer's performance dips after a peak since it maintains larger queues at higher capacities, eventually causing timeouts.

a valid solution to the Equation (16). Hence we conclude that $\{x^*_p : p \in \mathcal{P}\}$ and $\{\lambda^*_{u,v} : (u,v) \in E\}$, $\{\mu^*_{u,v} : (u,v) \in E\}$ are optimal primal and dual variables, respectively, for the optimization problem. The equilibrium rates found by Spider for the parallel network topology are optimal.

C Additional Results

C.1 Comparison with Celer

We run five circulation traffic matrices for 610s on a scale free topology with 10 nodes and 25 edges to compare Spider to Celer [11], a back-pressure based routing scheme. Each node sends 30 txns/s and we vary the mean channel size from 200€ to 6400 €. We measure the average success ratio and success volume for transactions in the 400-600s interval and observe that Spider outperforms Celer at all channel sizes. Celer splits transactions into transaction-units at the source but does not source-route individual transaction-units. Instead, transaction-units for a destination are queued at individual routers and forwarded on the link with the maximum queue and imbalance gradient for that destination. This approach tries to maximize transaction-units in queues to improve network utilization. However, queued-up and in-flight units in PCNs hold up tokens in other parts of the network while they are in-flight waiting for acknowledgements, reducing its capacity. Celer transactions also use long paths, sometimes upto 18 edges in this network with 25 edges. Consequently, tokens in Celer spend few seconds in-flight in contrast to the hundreds of milliseconds with Spider. The time tokens spent in-flight also increases with channel size since Celer tries to maintain larger queues. Celer's performance dips once the in-flight time has increased to the point where transactions start timing out before they can be completed. Due to computational constraints associated with large queues, we do not run Celer on larger topologies.

C.2 Circulations on Synthetic Topologies

We run five circulation traffic matrices for 1010s on our three topologies with all channels having exactly the tokens denoted by the channel size. Fig. 22 shows that across all topologies,

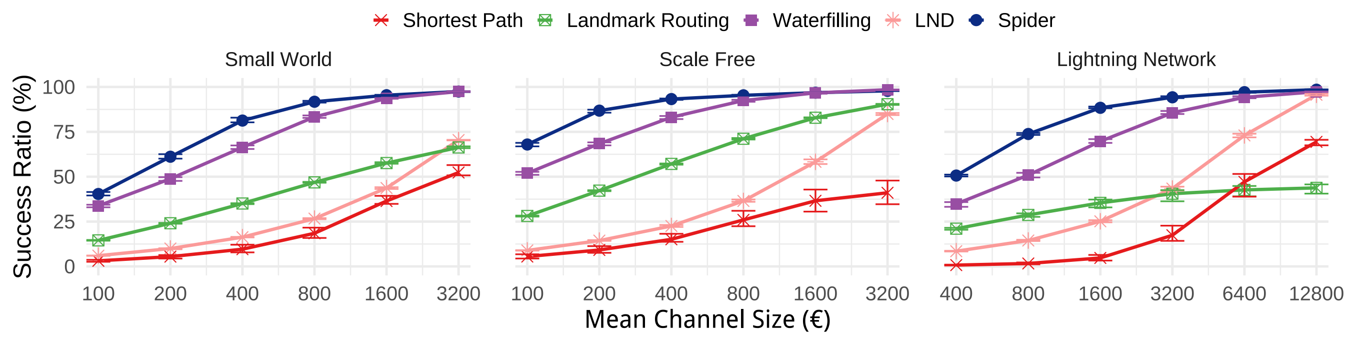


Figure 22: Performance of different algorithms on different topologies with equal channel sizes with different per sender transaction arrival rates. Spider consistently outperforms all other schems achieving near 100% average success ratio. Error-bars denote the maximum and minimum success ratio across five runs. Note the log scale of the x-axes.

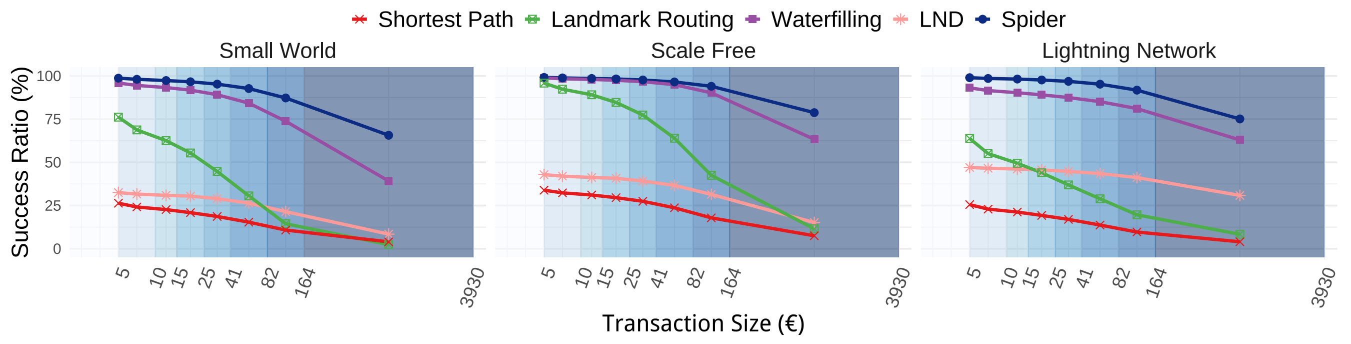


Figure 23: Breakdown of performance of different schemes by size of transactions completed. Each point reports the success ratio for transactions whose size belongs to the interval denoted by the shaded region. Each interval corresponds roughly to 12.5% of the CDF denoted in Fig. 7a. The graphs correspond to the (right) midpoints of the corresponding Lightning sampled channel sizes in Fig. 9.

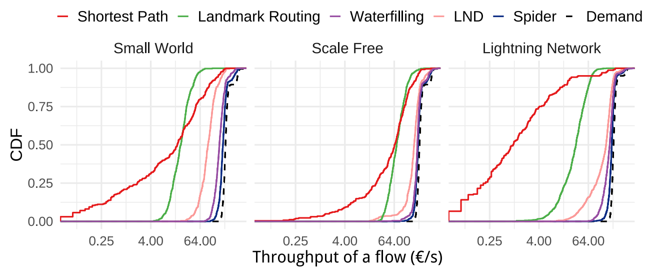


Figure 24: CDF of normalized throughput achieved by different flows under different schemes across topologies. Spider achieves close to 100% throughput given its proximity to the black demand line. Spider is more vertical line than LND because it is fairer: it doesn't hurt the throughput of smaller flows to attain good overall throughput.

Spider outperforms the state-of-the-art schemes on success ratio. Spider is able to successfully route more than 95% of the transactions with less than 25% of the capacity required by LND. Further Fig. 23 shows that Spider completes nearly 50% more of the largest 12.5% of the transactions attempted in the PCN across all three topologies. Even the waterfilling heuristic outperforms LND by 15-20% depending on the topology.

C.3 Fairness of Schemes

In §7.3, we show that Spider outperforms state-of-the art schemes on the success ratio achieved for a given channel capacity. Here, we break down the success volume by flows (sender-receiver pairs) to understand the fairness of the scheme to different pairs of nodes transacting on the PCN. Fig. 24 shows a CDF of the absolute throughput in €/s achieved by different protocols on a single circulation demand matrix when each sender sends an average of 30 tx/s. The mean channel sizes for the synthetic topologies and the real topologies with LCSD channel sizes are 4000€ and 16880€ respectively. We run each protocol for 1010s and measure the success volume for transactions arriving between 800-1000s. We make two observations: (a) Spider achieves close to 100% throughput in all three scenarios, (b)Spider is fairer to small flows (most vertical line) and doesn't hurt the smallest flows just to benefit on throughput. This is not as true for LND.

C.4 DAG Workload on Synthetic Topologies

Fig. 25 shows the effect of adding a DAG component to the transaction demand matrix on the synthetic small world and scale free topologies. We observe the success ratio and

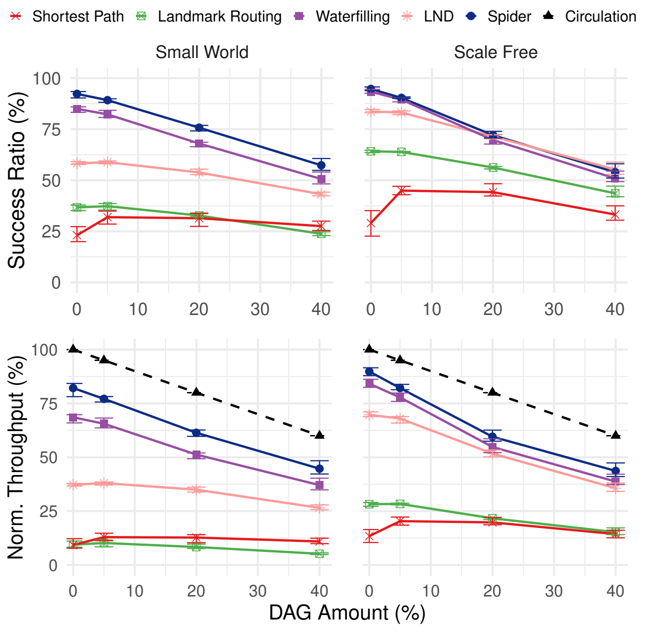


Figure 25: Performance of different algorithms across all topologies as the DAG component in the transaction demand matrix is varied. As the DAG amount is increased, the normalized throughput achieved is further away from the expected optimal circulation throughput. The gap is more pronounced on the real topology.

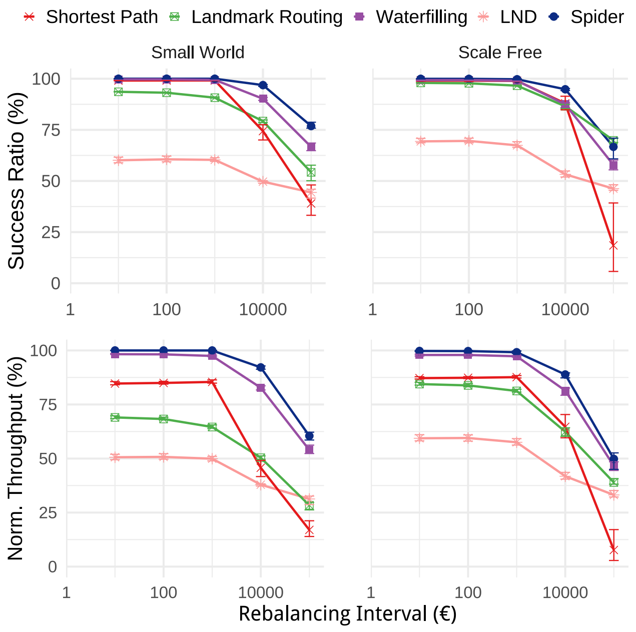


Figure 26: Performance of different algorithms across all topologies as the DAG component in the transaction demand matrix is varied. As the DAG amount is increased, the normalized throughput achieved is further away from the expected optimal circulation throughput. The gap is more pronounced on the real topology.

normalized throughput of different schemes with five different traffic matrices with 30 transactions per second per sender under 5%, 20%, 40% DAG components respectively. No scheme is able to achieve the maximum throughput. However, the achieved throughput is closer to the maximum when there is a smaller component of DAG in the demand matrix. This suggests again that the DAG affect PCN balances in a way that also prevents the circulation from going through. We investigate what could have caused this and how pro-active on-chain rebalancing could alleviate this in §7.4.

Fig. 26 shows the success ratio and normalized throughput achieved by different schemes when rebalancing is enabled for the 20% DAG traffic demand from Fig. 25. Spider is able to achieve over 95% success ratio and 90% normalized throughput even when its routers balance only every 10,000 € while LND is never able to sustain more than 75% success ratio even when rebalancing for every 10€ routed. This implies that Spider makes PCNs more economically viable for both routers locking up funds in payment channels and end-users routing via them since they need far fewer on-chain rebalancing events to sustain high throughput and earn routing fees.

PrivateEye: Scalable and Privacy-Preserving Compromise Detection in the Cloud

Behnaz Arzani[1], Selim Ciraci[2], Stefan Saroiu[1], Alec Wolman[1], Jack W. Stokes[1], Geoff Outhred[2], Lechao Diwu[2]

[1]*Microsoft Research,*[2]*Microsoft*

Abstract – Today, it is difficult for operators to detect compromised VMs in their data centers (DCs). Despite their benefits, the compromise detection systems operators offer are mostly unused. Operators are faced with a dilemma: allow VMs to remain unprotected, or mandate all customers use the compromise detection systems they provide. Neither is appealing: unprotected VMs can be used to attack other VMs. Many customers would view a mandate to use these detection systems as unacceptable due to privacy and performance concerns. Data from a production cloud show their compromise detection systems protect less than 5% of VMs.

PrivateEye is a scalable and privacy-preserving solution. It uses summaries of network traffic patterns obtained from the vSwitch, rather than installing binaries in customer VMs, introspection at the hypervisor, or packet captures. It addresses the challenge of protecting all VMs at DC-scale while preserving customer privacy and using low-signal data. We developed PrivateEye to meet the needs of production DCs. Evaluation on VMs of both internal and customer VM's shows it has an *area under the ROC curve* – the graph showing the model's true positive rate vs its false positive rate – of 0.96.

1 Introduction

Data center (DC) VMs today are largely unprotected – customers often don't use the compromise detection systems operators offer [1–3]. These systems monitor processes, network traffic, and CPU and disk usage from inside the VM or through introspection at the hypervisor to detect if the VM is compromised. We refer to them as OBDs (operator-provided and introspection-based detectors). Customers are reluctant to use OBDs due to privacy and performance concerns. Operators can mandate all 1^{st}-party VMs (those running the provider's workloads) use OBDs but can't require the same of their customers: our measurements of Azure reveal over 95% of VMs don't use OBDs! Operators are thus limited to using non-intrusive methods that prevent VMs from being compromised (e.g., firewalls, ACLs, [4, 5]). But these techniques do not always detect attacks before they succeed (see §2) and without additional protections, VMs in the DC can become and remain compromised for a long time.

It is important to close this gap and protect all VMs. Compromised VMs can be used to attack the DC infrastructure, or another customer's co-located VMs [6]. Operators need to be able to detect compromised VMs and protect all customers without needing their explicit permission or cooperation to do so. Our goal is to provide protection at *DC scale* while *preserving customer privacy*, *without visibility into customer VMs* and *without extensive and expensive monitoring*.

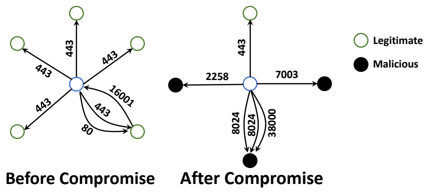


Figure 1: A VM's flows before and after compromise. The numbers on the edges are port numbers.

Here lies an interesting challenge: OBDs monitor process execution, check binaries and VM logins [1–3, 7–12], but without such detailed information, what hope do we have? Two observations help tackle this problem. First, we can learn the behavior of common attackers using information collected from VMs where OBDs *are* deployed. Second, a VM's flow patterns often change once it is compromised [13] e.g., the VM starts communicating with a command and control server (C&C), attempts to find and compromise other vulnerable VMs, or tries to attack the DC infrastructure. For example, Figure 1 shows a compromised VM discovered in Azure and its flow pattern before and after compromise. Our analysis of compromised VMs in Azure shows it is common to observe changes in network flow patterns when VMs are compromised. We expect our observations to be applicable to other providers' DCs as well.

Others have tried using network data to detect compromised machines but their work doesn't satisfy our scalability and privacy needs. Many studies [14–23] route traffic through middleboxes, rely on packet captures, or deep packet inspection (DPI) to extract features which are difficult, if not impossible, to gather through other means (e.g., packet payload). Continuous packet captures at scale come with prohibitive performance overheads and violate privacy by capturing application payloads. Packet captures contain personally identifiable information (PII), e.g. users' IP addresses, with stringent usage requirements [24]. Routing through middleboxes limits scalability, results in single points of failure, adds latency, and reduces throughput [25].

We present PrivateEye, a compromise detection system that runs at DC scale. PrivateEye is tailored to detect common attackers targeting the cloud, runs continuously, and complies with GDPR mandates [24]. It avoids expensive data collection by using flow pattern summaries to detect compromised VMs. It uses OBDs' detections on the VMs the operator can protect to learn the *change* in flow patterns of compromised VMs. Most OBD detections, which we use to train the model, are customer VMs (see §9), and so the learned model is expected to generalize to non-1^{st}-party VMs: PrivateEye applies this model to *all* VMs in the DC. PrivateEye uses random forests (RFs), an interpretable, supervised, machine learning (ML) model. These models generalize well and can infer, potentially complex, relationships in the input and are interpretable [26]. They often have higher accuracy compared to deployed heuristics (§4) while having similar performance overhead. PrivateEye leverages these properties to offer a practical solution for compromise detection at scale.

PrivateEye's role is to monitor all VMs in the DC and narrow the search space of VMs that need to be investigated. It is designed to have accuracy comparable to OBDs. Once PrivateEye identifies a suspicious VM, operators can use other, more invasive, methods which require customer permission to confirm its detections before shutting down the VM or moving them into a sandbox (see §5).

PrivateEye uses detections from OBDs inside VMs running real workloads to learn the change in the network behavior of compromised VMs. Deployed OBDs (only 5% of VMs) provide PrivateEye with a continuous feed of detections: PrivateEye can identify changes in the attacker's behavior as well as new attacks as it can be continuously retrained in the background. PrivateEye is one of the few systems that can leverage a continuous stream of detections. Attackers may attempt to avoid detection. Retraining may not be sufficient to detect all such attempts, but PrivateEye is still beneficial as it makes it harder for attackers to damage other VMs and the DC infrastructure: they would need to constantly modify their malware to avoid detection. Our contributions are:

1) Creating a scalable, privacy-preserving, compromise detection system that runs without needing customer permission. It operates without packet captures, without DPI, without fine-grained per-packet data, without using IP addresses, and without visibility into the VM.

2) Reporting on a deployment of PrivateEye's collection agent (CA) that has been running on *every* host across all DCs of our cloud for two years. It collects network-level data by querying the vSwitch [27] co-located on the same host.

3) Addressing the practical challenges of extracting features from coarse-grained flow summaries [28]. We use a novel feature construction approach that allows the ML model to detect changes in a VM's flow pattern while protecting customer privacy and keeping the feature vector small.

4) Evaluating PrivateEye using data from our public cloud as well as analyzing the model's false and true positives, feature importance, and design tradeoffs using the same dataset.

Our evaluations on a mix of both our internal and customer VMs, running real workloads, and set aside for testing, shows PrivateEye detects $\sim 96\%$ of the compromised VMs detected by OBDs with only a modest 1% false positive rate. This true/false positive rate is acceptable for our needs.

2 We need DC-scale compromise detection

We first show the need for DC-scale compromise detection:

Cloud VMs are constantly under attack. Brute-force attacks continue to pose a threat to DCs. We show the distribution of the arrival rate of SSH login attempts by unauthorized users to VMs located in 3 major cloud providers and 4 different regions across the globe (Figure 2). We deployed 120 VMs in each provider's DCs. We see VMs are subject to repeated login attempts and at least 3 VMs in each region experience at least one SSH login attempt per second (VMs in these experiments were monitored to ensure none of them are compromised). The time to discovery of a VM – the time from when it is deployed to the first SSH login attempt – is also short: many VMs are discovered in less than 15 minutes (Figure 3). VMs are under constant threat. This is not surprising but it serves to show we need constant monitoring of VMs in case any of these attempts succeed.

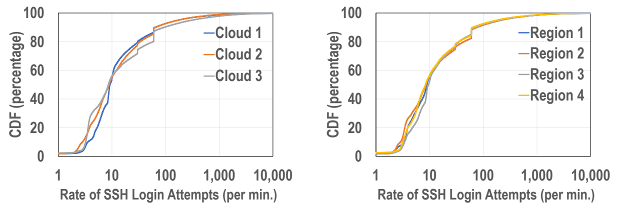


Figure 2: CDF of Rate of SSH login attempts for each provider (left) and for each region (right).

VMs *do* get compromised when customers are careless. Customers may fail to use strong passwords – providers enforce them, but many users change these passwords afterwards. Such VMs are susceptible to brute-forcers. We created 100 of them in our DCs – we ensured they were not co-located with other VMs and monitored them to ensure they did not harm other VMs. We chose passwords from the top 30 of the 1000 most used passwords [29]. The minimum time to compromise – the time from when it was instantiated to when the first successful login occurred – was 5 minutes (password 12345678) with a maximum of 47 hours (password: baseball). These VMs did not have OBDs, and none of them were flagged by any other intrusion detection service.

Compromised VMs are used to attack other VMs. Many exploits require code to be co-located with the victim. Access to a VM in the cloud allows attackers to bypass ACLs and firewalls that only protect VMs from external attackers. Compromised VMs may attempt to compromise other VMs: in one day, our OBDs found 1637 VMs attempting SQL-injection attacks and 74 attempting brute-force login. While small compared to the massive scale of the DC, these numbers only describe those VMs with OBDs. The magnitude of the problem is much greater when scaled up to all VMs.

OBDs (during Jan-June 2018) showed 14% of alerts were VMs brute-forcing other VMs and 13.87% were VMs scan-

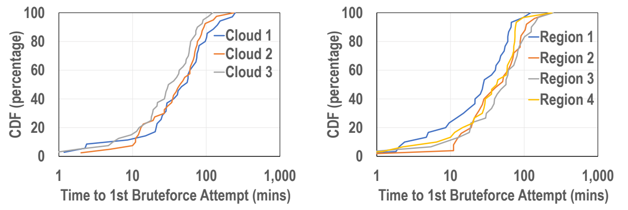


Figure 3: CDF of time between VM deployment and the 1^{st} login attempt per provider (left) and region (right).

	PrivateEye	Heuristic
AUC	0.96	0.5

Table 1: Comparison of PrivateEye to deployed heuristic.

ning for vulnerable ports. Furthermore, 7.7% of compromised VMs were compromised through brute-force attacks.

3 PrivateEye's threat model

Our threat model is similar to other infrastructure services. We do not trust any VM (they can run arbitrary code). We assume we can trust the hypervisors, network, and hardware.

PrivateEye relies on detections from OBDs deployed on a subset of VMs. We use this data as labels during training for the ML model. We assume this data cannot be faked, manipulated, or altered. We also assume OBDs can accurately detect when a VM is compromised. Specifically, we assume false positive/negatives rate of OBDs is sufficiently small – after all, they are accurate enough to be used to protect the provider's 1^{st}-party VMs which are critical to its business.

Malware can adapt its behavior in order to conceal malicious behavior. We assume OBDs adapt to such changes and re-training PrivateEye with their more recent detections can help it adapt to them as well (see §9). We assume many attackers do not discriminate between VMs that are protected by OBDs and those that are not: the same attackers that successfully compromise protected VMs can also (one can argue more easily) compromise others and by learning their behavior, through OBD detections, PrivateEye can protect other (unprotected) VMs in the DC from these attackers.

4 Simple heuristics are ineffective

Our operators have used insights from past OBD detections to build a rule-based solution. To motivate using ML, we compare PrivateEye to this strawman which was used to protect VMs without OBDs in our DCs. It encodes learnings from honeypots and past OBD detections to a per-VM score which measures the similarity of a VM's flows to those attacking honeypots or past OBD detections. Metrics such as having more than 500 flows/sec to a port/IP, changing DNS servers, or too many DNS flows increase the score. The increase is weighted by the operators' confidence in the signature.

This (strawman) heuristic identifies VMs engaged in brute-force or port-sweeping attacks accurately but rarely detects other compromises. We use the area under the receiver operating characteristic curve (ROC) – the graph that plots the true positive rate vs false positive rate of an algorithm – or AUC to measure accuracy. Higher AUCs indicate better accuracy. The heuristic has an AUC of 0.5 (PrivateEye's is 0.96). Indeed, by only testing on port-sweeping VMs, the heuristic's AUC increases to 0.69. Looking at its false positives, 10 legitimate VMs, spanning 3 Virtual Networks (VNets), had scores above 10 (A VNet is a virtual network set up by a user to connect its VMs). Three of the VMs were in our canary VNet. Canaries continuously ping on port 10000 to check network connectivity: all of the VNet's VMs had many flows to port 10000 which is why they all had high scores

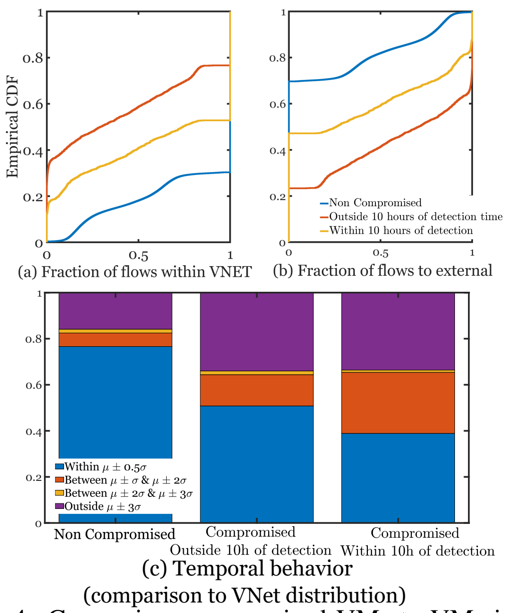


Figure 4: Comparing compromised VMs to VMs in their VNET. μ: average Bps of a VNet to each destination; σ: standard deviation. (a) Distribution of flows to VMs in the VNET. (b) Distribution of flows to IPs outside of the DC. (c) Temporal behavior.

(≥ 2). PrivateEye's high AUC shows using ML with the right features helps avoid such false positives.

5 PrivateEye's design requirements

Our design requirements for PrivateEye are:

GDPR compliance. The European law on data privacy (GDPR [24]) mandates any personally identifiable information (PII) should be tracked in case the customer wishes to inspect (or delete) it. Operators are required to answer customer requests within 48 hours and have two choices: join and tag data with meta-data to be able to identify which customer it relates to or avoid storing any and all PII data. For example, a VM's public IP (and the IPs it communicates with) has to be mapped to the customer's account and stored with *all* network telemetry from their VM. Public IPs are dynamically allocated and would need to be tracked in time which can result in significant and unnecessary overhead.

Even without such customer requests, this data has to be deleted from all company data-stores after 30 days to avoid violations. GDPR makes it expensive to sustain solutions relying on PII data. PrivateEye relies on 10-minute flow pattern summaries and ignores specific IP addresses.

Low runtime overhead. PrivateEye should have low performance overhead, should be able to run at DC-scale, and shouldn't interfere with ongoing traffic. Many, state-of-the-art, compromise detection systems have high performance overhead and cannot be used extensively in the cloud. For example, DPI (e.g., [15]) adds additional per-packet delays which prohibits serving traffic at line-rate ($40 - 100$ Gbps). Packets may be mirrored to dedicated middleboxes that can run DPI off the critical path but our experience with simi-

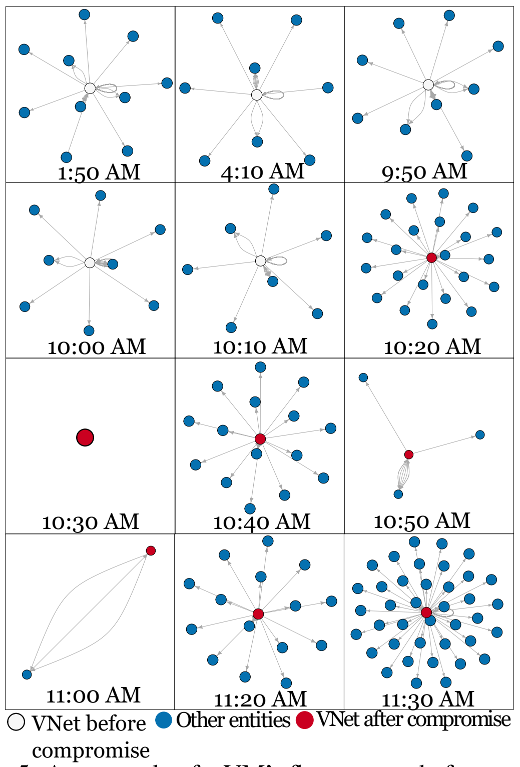


Figure 5: An example of a VM's flow pattern before and after it was compromised.

lar systems (e.g., EverFlow [30]) show they put significant load on the network and that they are hard to scale. We can re-implement DPI based solutions using new programmable switches [31] to improve performance, however, these solutions are not yet ready for production as packets need to be re-circulated [32] which prevents serving packets at line-rate.

Detect malware in the wild. Many past approaches observed malware in a sandbox to build behavioral signatures (e.g., [7]). Malware may change its behavior if it detects it is in a sandbox (e.g, [33]). OBD detections from production VMs provide a rich dataset about malware behavior in the wild which PrivateEye can learn from. PrivateEye uses this learned behavior to detect other instances of compromise.

Ability to generalize. Customers constantly bring up new VMs and shut-down old ones. We should not rely on learning from specific VMs or even customers. PrivateEye can generalize to VMs (and even customers) not in its training set: our evelauation test sets (see 8) comprises of customer and internal VMs and VNets that are never in the training set.

Operate as a first line of defense. OBDs use extensive monitoring. PrivateEye has a more restricted view of the VMs. It is used as a preliminary detector to avoid unnecessary penalty (OBD mandate) on a large number of customers. Its goal is to reduce monitoring overhead and operational complexity and to protect *all* VMs without needing customer permission until further investigation is necessary. Once it flags a VM as suspect, it can raise an alert to the customer to ask for permission to investigate the VMs further through other more invasive and expensive techniques at the operator's disposal.

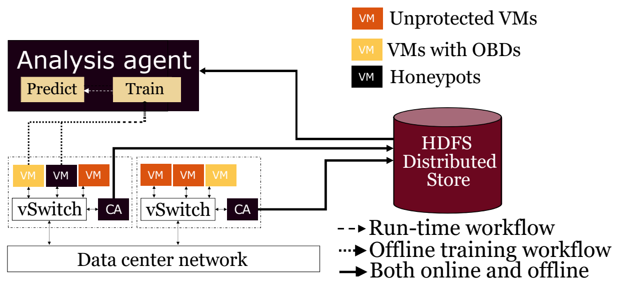


Figure 6: **System overview of PrivateEye.**

For example, our operators have access to VHD-scanners and can also use DPI-based systems on *one-time* packet captures of the VM's traffic[1]. PrivateEye provides "just-cause" for operators to use these tools when it flags a VM as suspect. These approaches are automated and can be run without operator intervention. PrivateEye assigns scores to each detection allowing operators to pick the right tradeoff between the true and false positive rates for their needs. PrivateEye is not meant to fully replace OBDs, we encourage customers who require stronger protections to opt-in to OBDs providers offer.

6 System Design

PrivateEye runs continuously and scales to large clouds with low overhead. The privacy sensitive fields it collects are anonymized using a keyed hash message authentication code (HMAC) during data collection and *deleted* once we construct the features. Figure 6 shows PrivateEye's design. We use two arrow types to differentiate training and run-time workflows. PrivateEye has two parts: the collection agent (CA) and the analysis agent (AA). The CA is responsible for data collection and the AA for analysis and detection. We next describe the key ideas behind its design:

A VM's flow pattern changes when it is compromised. We have observed a VM's flow patterns change once compromised. Almost all malware we studied (using our honeypots) changed the machine's DNS, few connected to the same set of external IPs, some connected on the reserved port for NTP to non-NTP servers, and those mining for digital currency had flows on port 30303. We ask whether these changes are visible on VMs running real workloads (as opposed to idle honeypots)? We leverage 1-month of our OBD detections to answer this question and compare the flow pattern of compromised VMs to others in their VNet. Within each VNet, we see similar behavior for all VMs. But when the VM is compromised, it starts to deviate from the typical behavior of other VMs in its VNet (Figure 4).

Figure 4 a-b compares the *spatial* distribution – the fraction of flows going to other VMs belonging to the same customer vs. to other VMs in the DC vs. machines outside of the DC – of flows originating from VMs sharing a VNet. We observe only 30% of the non-compromised VMs have flows destined to IPs outside of their VNet whereas this number is as high as 50% as we get closer to when the compromise was detected and roughly 80% around the time of detection. It

[1]These are only collected *if* the VM is suspected.

seems, at least in these instances, after the VMs were compromised they tended to communicate more with destinations outside their VNet. The VM's *temporal* behavior also shows changes in behavior around the time of compromise (Figure 4-c). The volume of traffic each VM sends to *individual IPs* is often close to the VNet average but when compromised, VMs exhibit increased deviations from this mean. Our manual analysis of their flow patterns showed noticable changes in behavior after compromise. We omit most of this analysis due to space restrictions but show one example. We show (Figure 5) the flow pattern of a 1^{st}-party VM before it was detected as being compromised (11:30 AM). We see the VM's flow pattern change drastically from its, previously stable, normal behavior when we get closer to the time of detection. We use these insights when constructing features.

We can get accurate flow-summaries from the vSwitch with low overhead. The vSwitch [27] processes all packets of all flows to/from the VM and keeps simple, per-flow state (Table 2) for all active flows. PrivateEye leverages this feature to obtain accurate per-VM flow summaries (see §7.1).

Using OBDs to create labeled data for training. PrivateEye is trained on data from VMs running OBDs. Our DCs run two types of OBDs: (a) Those running on our 1^{st}-party VMs: these VMs often have in-kernel instrumentation for detecting compromise. (b) Those running on customer VMs: customers can opt-in and use OBDs and if they do so, they can deploy and run them inside their VMs. All of the OBDs we use are built on top of Defender [34] (Windows) and ClamAV [35] (Linux) but also monitor irregular login behavior, system calls, and many other parameters. Although OBDs are intrusive the system as a whole meets our privacy requrements: Azure owns all 1^{st}-party VMs, and the only 3^{rd}-party VMs where we use OBDs are those where the customer has explicitly given permission. Note, customer OBDs are less common but, interestingly, despite their lower popularity most compromise detections are from these OBDs (see §9).

We hypothesize many compromised VMs exhibit similar flow-pattern changes to those compromised VMs that were detected by OBDs; because often attackers run attacks against IPs in the cloud irrespective of the services deployed behind those IPs or who owns them (non-targetted attacks §2). Our evaluations confirm this hypothesis as PrivateEye is tested on internal and customer VMs and workloads that are not in the training set and achieves an AUC of 0.96.

Using supervised-learning to learn flow-pattern changes. Anomaly detection and clustering approaches seem natural approaches for solving our problem. We have tried anomaly detection [36], cross entropy [37], ECP [38], TSNE [39], and k-means [40], and also AutoEncoders [41] but anomalies were routinely observed in many VM's lifetimes and it was hard for operators to distinguish between anomalies that were caused by malware and those which were intended VM behavior. Even a tainting + clustering approach i.e., marking points in the clusters with compromised examples as compromised, resulted in higher false positives compared to PrivateEye.

Metric	Description
Time	Timestamp of data collected
Direction	Incoming to/Outgoing from VM
Anonymized Source IP	Source IP in first SYN packet*
Anonymized Source VNetId	VNetId of source IP if any*
Anonymized Dest IP	Destination IP in first SYN packet*
Anonymized Dest VNetId	VNetId of destination IP if any*
Protocol	Protocol if known
Dest Port	Destination port in first SYN packet
BPS In	# of bytes/sec incoming to VM
BPS Out	# of bytes/sec outgoing from VM
PPS In	# of packets/sec incoming to VM
PPS Out	# of packets/sec outgoing to VM
Unique Flows	# of unique 5-tuples collected
Total Flows	# of unique 5-tuples (both collected and missed flows)

Table 2: Data collected by the CA (*not* the features). *These fields are removed entirely after feature creation.

We need to detect *specific changes* that point to the VM becoming compromised (as opposed to finding all anomalies). We chose supervised learning and specifically random forests as they have low overhead. They are debug-able, explainable, highly accurate, and resilient to overfitting [42]. They construct multiple decision trees over a random subset of features during training. Each decision tree uses a greedy algorithm to (1) iteratively pick features with the most information gain [43], (2) makes a decision using values of each feature, and (3) iterates until it reaches a "leaf". Leaves either consist of samples with a single label, or have samples where one label is the majority. At run-time, the algorithm traverses the tree for each test case and returns the majority label (from training) at the leaf. Random forests output the mean prediction across trees and the fraction of trees that predicted each label. We use this fraction as a score to measure confidence and to control PrivateEye's false positives. Operators use it to decide what to do. We set the model's hyperparameters (e.g., max-depth) using Bayesian optimization [44].

Using informative features. In section §7.1 we will describe PrivateEye's CA and how it collects the raw data in Table 2. Privacy-sensitive fields, such as the IPs and VNet ID, are anonymized. Here, we describe the raw data itself, the challenges in extracting privacy-preserving features from this raw data, and how we create these features.

PrivateEye uses three sets of features: graph-based, protocol-based, and aggregate features:

(1) Graph-based features. We want to detect the changes in a VM's flow pattern that indicate it is compromised. The IPs the VM connects to are a crucial part of these flow patterns but using them as features is not possible for two reasons:

Privacy – IPs are anonymized and removed once the features are created. We cannot map IPs geographically nor can we classify them according to the AS that owns them.

Data-sparsity – using IPs as features results in a large feature vector (2^{32}) and by extension an extremely sparse training set. The curse of dimensionality dictates: to maintain accuracy, as the number of features increase, the number of training samples must also increase [45]. This is especially problematic for training supervised models as compromised

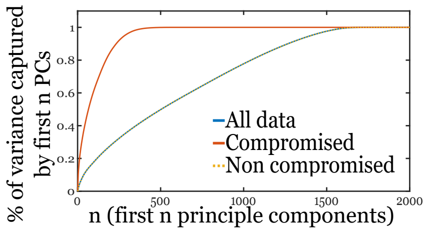


Figure 7: Variance captured by the first n PCs.

VMs are rare – our datasets are imbalanced and have more non-compromised examples and far fewer compromised examples. Using IP-prefixes is not possible: the smallest usable prefix size (to avoid spanning multiple ASes) is \24 but this results in a 2^{24} feature vector which is, again, too large.

Proior work [20, 28, 46, 47] have acknowledged this problem and attempted to solve it by operating at the granularity of flows instead of VMs – classifying individual flows as malicious to avoid using IPs as features. While finding malicous flows is useful when implementing ACLs or firewalls it is hard to identify compromised VMs without viewing their flow patterns as a whole and, in some cases, comparing the VM's flow patterns to that of others. Other works in networking (e.g., [42, 48, 49]) have also used ML. But IPs are not relevant to the problems they tackle and are not used as features. But in the context of compromise detection, IPs play an important part. We present a novel feature construction approach which allows us to summarize graph evolutions, including IP-related changes, using temporal and spatial features. We do not need access to the specific IPs for constructing these features (avoiding privacy problems) nor do we need to use 2^{32} feature-vectors (avoiding data-sparsity problems). Our intuition, based on the observations we presented earlier, is that to detect changes in a VM's flow pattern, features should describe its change both in time and space and compare it not just to its own past behavior but also to others.

Temporal graph-features: capture how a VM's flow patterns change over time compared to itself and other VMs. We will first describe the intuition behind our solution in the context of an example:

Suppose a compromised VM connects to a C&C server with IP a.b.c.d. We can, for all VMs in the DC, build the CDF of bytes sent to IP a.b.c.d over time. If only a few of the other VMs in its DC have been compromised, flows to IP a.b.c.d from the compromised VM would fall in the top portion of this CDF, e.g. the top 10%. The flow to IP a.b.c.d falls into this top 10% interval because such flows were unique to compromised VMs – by mapping the flow to this interval we capture the relevant information for detecting whether the VM is compromised. Other, similar changes are also possible. This is why we use a combination of four such CDFs:

- For each VM, the distribution of each flow according to its "Bps out" over the course of one hour.
- For each VM, the distribution of all flow according to their "Bps out" over the course of 10 minutes.
- For each remote IP, the distribution of all flows to that IP address across all VMs in the DC according to their "Bps out" over 10 minutes.
- For each remote IP, the distribution of all flows connecting to that IP address across all VMs in the DC according to their "Bps out" in 1 hour.

We divide each CDF into five buckets: top 1%, top 1-10%, middle, bottom 1-10%, and bottom 1%. Flows are then projected to a lower dimension based on the bucket they fall into on each of these CDFs . The results are then combined with other features for each VM in each 10 minute period.

These CDFs describe a VM's flows through time as compared to itself and other VMs in the same region. There are other possible CDFs we could use. Finding the optimal selection is the subject of future work.

Spatial graph-features: We classify each flow in one of three categories based on its endpoints: (a) both are in the same VNet (b) both belong to the cloud provider (different VNets) (c) one is an external IP. We aggregate each metric in Table 2 for each group. We can build these groups using the anonymized VNetIds (which we remove after feature construction). Specifically, the same anonymized VNetIds point to VMs in the same VNet, different ones point to VMs in different VNets, and absent Ids point to external IPs.

(2) Protocol Features. A flow's destination port can help identify the application protocol being used. Often attackers/malware use specific protocols for communication. There are numerous examples we found when deploying PrivateEye where the set of protocols used by the VM changed once it was compromised. We created a list of 32 ports of interest including 22 (SSH), 53 (DNS), 80 (HTTP), 123 (NTP), 443 (TLS), 30303 (mining digital currency), and 3389 (RDP). For each of these ports, we aggregated five of the metrics shown in Table 2 to construct six features: "Direction" (incoming vs. outgoing), "PPS In", "PPS Out", "BPS In", "Unique Flows".

(3) Aggregate Features. Finally, we also use the total number of bytes sent/received and the total number of incoming/outgoing connections as additional features.

There is no good linear summary of the feature-set. We have constructed $k = 2116$ features. We next check whether there is a more compact representation of the data using Principle Component Analysis (PCA) [50]. Each principal component (PC) corresponds to an Eigenvalue of the data, and the sum of these Eigenvalues equals its variance. We find we need to keep 75% of the PCs to keep 99% of the variance (Figure 7). Therefore, PCA is not a good candidate for dimensionality reduction in this problem.

Interestingly, if we only focus on compromised VMs, we see 99% of the variance in the dataset can be captured with only 16% of the PCs: the space of compromised VMs (as described by these features) is more compact. But, there is little overall linear dependence across features. This is yet another motivation for using information theoretic and/or ML-based techniques as the number of features is large and it is hard to build human-tuned heuristics using these features.

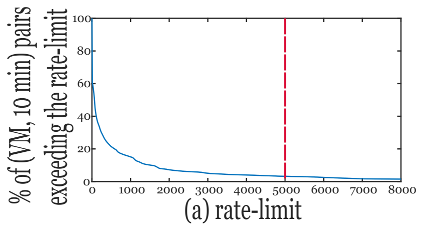


Figure 8: Percentage of (VM, 10 minute) pairs (y-axis) with more than n (x-axis) flows.

7 System Implementation

We next discuss each of PrivateEye's building blocks in more detail and describe how they are implemented in practice.

7.1 The collection agent

A highly performant CA is crucial in achieving our performance and scalability requirements. We have deployed our CA on all hosts across all our DCs. It runs continuously and polls the vSwitch for the data in Table 2 every 10 seconds. The vSwitch records this data for each flow and for each VM within this period: the choice of polling period does not result in data loss but only impacts CPU usage.

The interface to the vSwitch uses read locks[2]. Although the vSwitch has higher priority to obtain the lock, the CA limits its query to 5000, randomly selected, flows per 10s and per VM to reduce the impact of contention. If the total flows within the polling period exceed this limit for a VM, the vSwitch reports the total. To reduce the overhead of saving data, the CA aggregates some of the fields in Table 2 over 10-minute epochs. This aggregation is on the fields that store bytes, packets, and flows, and we also aggregate flows based on their destination port. As a result, a 10-minute dataset from a region, with over $300,000$ servers, is 109 MB.

To choose the 5,000 limit we looked at the 10s epochs with more flows than any given limit, for one hour, in one DC (Figure 8). Only 4% of samples have more than 5,000 flows. Thus, using this limit results in data-loss for only 4% of the samples. Our data shows on average VMs have 1843.9 ± 9.5 flows in each epoch. The distribution has a long tail; when the number of flows exceeds 5,000 the number of flows is 18890.6 ± 104.0. We accept this loss and show in §8 that we can detect compromised VMs despite this limit. The vSwitch team confirmed the vSwitch continued to process packets at line-rate when using this limit.

We designed the CA from scratch despite systems such as NetFlow [51] and IPFix [52], which are already deployed in our DCs. These systems are used for traffic engineering, DDOS protection, and other tasks. They run on our core routers and sample 1 out of 4096 packets traversing the network core. Because of this sampling, they are biased towards monitoring "chatty" VMs and "elephant" flows. Also, they do not capture flows that do not traverse the network core. Therefore, IpFix/NetFlow are not adequate for PrivateEye which requires more complete knowledge of per-VM flow patterns. Work such as [53] show the shortcomings of such monitoring systems even when used in heavy-hitter detection systems. Other in-network monitoring systems such as [4, 54] are also inadequate as they require a specific type of sampling unsuitable for PrivateEye [4] or require hardware upgrades currently not possible [54]. PrivateEye limits the number of records it extracts in each 10s. This limit also results in occasional data-loss but does not suffer from the same problems. The limit is applied to each VM separately and doesn't bias the dataset towards chatty VMs. We capture data from the host's vSwitch which has all records for the VM's flows irrespective of where they are routed. The CA is a software component on the host and requires no hardware changes. IPFix captures fewer flows per VM than our CA and also misses capturing traffic from a large number of VMs (Figure 9).

[2]The table is also used by other systems that need a consistent view.

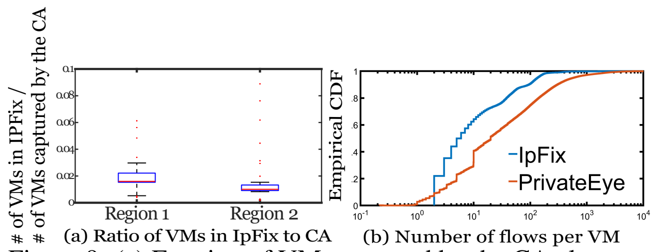


Figure 9: (a) Fraction of VMs captured by the CA also captured by IpFix. (b) CDF of number of flows captured per (VM,10 min) for IpFix vs. the CA (x-axis log scale).

The CA has low overhead – typical usage of only 0.1% of the host CPU and a maximum of 15 MB of RAM.

Finally, we note the CA can also be implemented using programmable switches. We use vSwitches as they are more widely deployed in our networks (and those of others).

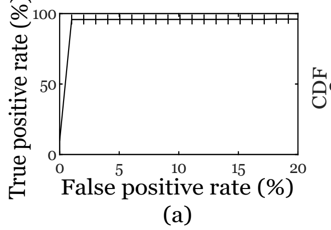
7.2 The Analysis Agent

The AA has two roles: (1) train a classifier using past detections of OBDs (offline), (2) to run the classifier and predict which VMs are compromised (online). It is alerted when there is a new detection from any of the OBDs on any of the VMs and tags the data from those VMs with a "compromised" label. This data is then added to the training set. This training set is persisted in a distributed data store. The AA is periodically retrained using this data to keep up with any changes in the set of malware or the OBDs themselves.

Data collected by the CA for all other VMs (those without OBDs) is sent through a stream processing pipeline where we create the features. These features are then passed through the RF model which determines whether the VM is compromised. We are in the process of deploying the AA in our DCs. We describe this deployment in more detail in §9.

8 Evaluation

We evaluate PrivateEye using data from Microsoft's cloud. We have shown parts of the evaluation – which helped justify our design choices – in earlier sections (e.g.,§7.1). In this section, our goal is to answer questions about PrivateEye's accuracy (§8.2), the features that help it achieve this accuracy (§8.2), the causes behind its mis-classifications (§8.2), and its performance overhead (§8.3). We also looked into how it can



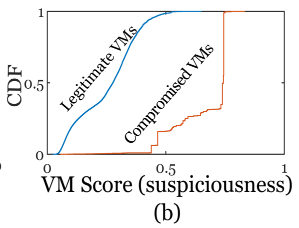


Figure 10: (a) PrivateEye's ROC. (b) PrivateEye's scores (CDF).

be used in the context of an example use-case (Appendix §A).

8.1 Methodology

Data. Each point, (x_i, y_i), in our data corresponds to a VM in a 10 minute period, where x_i is the 2116×1 feature vector described in §6 and y_i is a label: compromised or legitimate.

Labeling. We use OBDs from over 1,000,000 internal and customer VMs to create labeled data for evaluation. These VMs run diverse workloads and use both Windows and Linux OSes. A direct implication of this labeling is that PrivateEye's accuracy can only be as high as OBDs. OBDs can have false positives (negatives), but because operators use them to protect their own 1^{st}-party VMs, we assume these are negligible. OBDs with higher accuracy only improve PrivateEye.

We train and test PrivateEye on *all* detections from these OBDs – we do not restrict PrivateEye to detecting a particular type of compromise: its goal is to detect any compromise OBDs can detect. Most OBD detections were from (different) *customer* VMs (0.87 of the total compromises) – see §9 for further discussion on this. For most OBD detections we also saw external, network-level, signs of malicious behavior. When available, we also report on results from the operator's manual investigations to confirm their detections.

Train-test split. We need to split the data into a train and test set. It is important to do so correctly: if there is *information leakage* between the train and test set we may artificially boost accuracy. For example, we cannot split the data by time because some VMs will have lifetimes spanning both the train and test set. In training, the model may learn the behavior of the VM itself as opposed to whether it is compromised, and when used in practice, it would have much lower accuracy than what we see in testing. The test set should be representative of how the system is used in practice – it is tested on VMs it has never seen before (VMs that are not in the training set). These constraints imply the VMs in the training set cannot be in the test set. We take an even more conservative approach: *We split the data by VNet*. This ensures each VM, and those in its VNet that have similar workloads (§2), only appear in either the training or the test set but not in both.

Addressing class-imbalance. RFs need a similar number of training samples for each label to achieve high accuracy. Unfortunately, our data suffers from *class-imbalance*: a small fraction of our data is labeled compromised (only 0.1%). This can lead to an RF that classifies everything as legitimate while achieving (artificial) high accuracy. We use a standard technique to solve this issue: down-sampling [55]. Down-

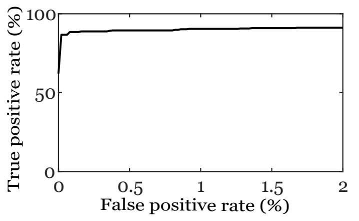


Figure 11: ROC for per VM classification.

sampling randomly chooses a subset of the more popular label for training. But too much down-sampling can reduce the volume of data available for training which can also hurt accuracy. We use Bayesian Optimization to find the right trade-off. To improve accuracy, we use the 'balanced' option in SKLearn's RF function which weights the samples for each label to make up for the lower number of samples.

Performance metrics. PrivateEye's output for each sample x_i at time t_i is a *score* and a decision on whether the VM was compromised in the 10 minute epoch prior to t_i. The score measures the level of suspicion towards that VM. Operators can use it to decide whether to investigate the VM further.

We use an ROC to measure accuracy – we vary the threshold applied to the scores to decide whether the VM is compromised: ROC shows the true positive rate (TPR) vs the false positive rate (FPR). The area under the ROC (AUC) summarizes the significance of the ROC: an AUC of 1 indicates a perfect test, and an AUC of 0.5 indicates a random test [56].

8.2 Classifier Evaluation

Our goal in this section is to answer questions such as:

- What is PrivateEye's accuracy?
- What causes PrivateEye's false positives/negatives?
- How does PrivateEye's RF compare to other models?
- Are the features we created effective?
- How does PrivateEye's sampling limit affect accuracy?

We next describe the answer to these questions in detail.

Accuracy: PrivateEye has a high AUC (0.96). We use 20-fold cross validation and show the mean of the 20 outputs and the 95^{th} percentile confidence interval (Figure 10-a). PrivateEye can detect 95.77% of compromised samples with 1% FPR. The scores are correlated with the labels (Figure 10-b): the gap between the distribution of scores for compromised and legitimate VMs confirms the accuracy of the model.

PrivateEye's long-term accuracy is also high. PrivateEye checks each VM every 10 minutes to see if any are compromised. But what if operators want to do so less-often e.g., before the VM is shut-down (once over the VM's life-time)? They can aggregate the scores across consecutive samples to do so. Many aggregators are possible: we use the sum of all scores over the lifetime of the VM (other aggregators achieved similar results). The ROC is produced using this new score (Figure 11). PrivateEye's accuracy remains high (90.75% TPR for 1% FPR) albeit slightly lower than before – possibly due to the longer lifespan of legitimate VMs: compromised VMs are typically shut-down more quickly resulting in a lower aggregated score. This result also confirms the ROC in

Figure 10-a is not dominated by samples from a single VM.

We further experimented with different choices of n, where n is the number of 10-minute intervals that need to pass before we make a prediction. The ROC curves were bounded by those of Figure 10-a and b but do not show an explicit trend.

PrivateEye's FNs were *mostly* OBD false positives. PrivateEye's FNR (average) is 5%. This implies (on average) 5% of samples were reported incorrectly as legitimate. But most (70%) of these samples were actually false positives (FPs) of the OBDs (remember OBDs tend to be conservative as they protect internal VMs) – operators investigating the alerts identified them as FPs e.g, in one case the investigation report mentioned: "This was a FP detection as the vulnerability scanner contains data to scan for CVE-2006-3439".

Our investigations of PrivateEye's FNs revealed other interesting information. For example, we grouped FN samples based on which VM they describe and found 86% were cases where all samples for the VM were labeled legitimate. These VMs are part of the training set for other folds during cross validation: it appears PrivateEye is resilient to errors in the labels it uses for training (given the 95% TPR). We need to investigate this point further to confirm this hypothesis.

Some VMs had a mix of compromised and legitimate detections (14% of FN samples). These VMs were involved in port-sweeping attacks but we found no correlation with the number of active flows or the volume of traffic. OBDs assign a "severity" to their detections to report their confidence in the detection. All of PrivateEye's FNs were low severity.

PrivateEye's FPs have similar flow-patterns to compromised VMs. We manually inspect the flow patterns of VMs PrivateEye mistakenly reported as compromised. Most such VMs had a small number of flows to non-reserved ports. In a few cases, the VM had *only* a small number of flows on the DNS port (53) to another VM in its VNet (VMs in the same VNet belong to the same customer). In another instance, the VM had *no* outgoing flows, but multiple VMs from the DC were attempting RDP connections to it. None of the VMs in these VMs' VNets had similar flow patterns. These behaviors are similar to compromised VMs which explains why these VMs were mistakenly flagged by PrivateEye.

PrivateEye detected attacks OBDs missed. We observed two separate instances where SQL servers were conducting port-sweeping attacks on another VM. In both cases the attack lasted for one 10 minute epoch but PrivateEye detected it. OBDs did not. These VMs were only active for a short duration (less than a few hours). The short period of the attack and the short life-span of the VM may explain why the

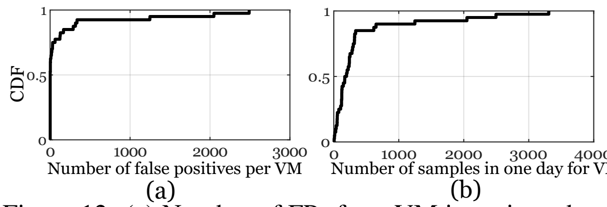


Figure 12: (a) Number of FPs for a VM in a given day. (b) Number of samples in a day for VMs with an FP.

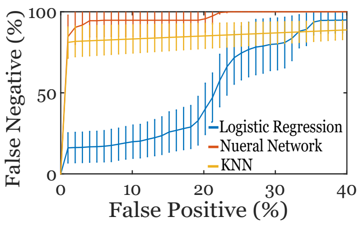


Figure 13: Performance of other ML algorithms.

OBDs did not detect these attacks.

Consecutive detections can help reduce FPs. We ran samples for legitimate VMs in the test set through PrivateEye and grouped the results by VM. For over 60% of these VMs PrivateEye had only a single FP (Figure 12-a): we could use consecutive detections as a potential means of reducing FPs. For example, we can change the detection granularity to every 20 minutes instead of 10 where we require two consecutive detections to declare a VM compromised. This approach can result in reduced true positives – as seen earlier in the more extreme example where we make one decision in the entire lifetime of the VM. Further understanding of PrivateEye's FPs requires manual inspection from within the VM but, sadly, we are unable to report on the results of such analysis.

Random forests (RFs) are a good first choice [14, 48, 57]. We compared RFs with many other ML algorithms and show RoCs for a subset (Figure 13). RFs outperformed all other models we tried. The closest algorithm to the RF were neural networks[3] (NN) which have an 81% TPR for a 1% FPR.

All features contribute to the detection. We have seen there is no compact, linear, representation of the features that would capture all the information in the data §6. We dig deeper to see which class of features are most helpful. To do so: (1) we use each class individually (Figure 14-a), and (2) we remove each class altogether (Figure 14-b). Removing graph features individually (spatial, temporal) has little impact on TPR (0.2%) but removing both can drastically reduce it (18%). Most classes (except spatial features) can find compromised VMs with a TPR $\geq 70\%$ and an FPR $\leq 5\%$. We conclude all features significantly contribute to detection (though they are not equally important). To validate these observations, we experimented with various feature selection techniques (Figure 15). We refer the reader to [58] for the description of these techniques due to space restrictions.

Aggregate features have good predictive power: a TPR of 77.7% for 5% FPR when used as the only features and resulting in 7% drop in TPR when removed: were most compromised VMs engaging in volumetric attacks? We found this *not to be the case* – the maximum traffic sent by compromised VMs across all samples was 2.6 MBps (median 0.0 Bps) vs. 10.3 GBps (meidan 189.62 Bps) for legitimate VMs.

Comparison to other VMs helps detection. Graph features compare the VM to others by mapping flows onto intervals on various CDFs (see §7.2). What happens if we use CDFs that

[3]We used a single hidden layer with 1000 neurons. Experiments with additional hidden layers in the NN and different numbers of hidden dimensions produced similar results.

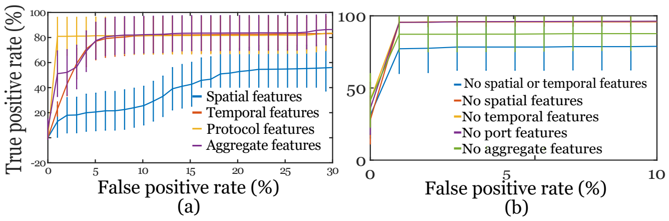


Figure 14: (a) Contribution of each feature class to the overall accuracy. (b) ROC without each feature class.

compare a VM with its own history instead of that of all VMs in the DC? The TPR drops to 80% for 1% FPR (Figure 16) indicating the comparison to other VMs is indeed useful.

The choice of sampling limit is important. The CA rate-limits the number of flow entries it queries from the vSwitch to $5,000$ every 10 seconds. This rate-limit allows us to capture over 97% of each VM's flows (§7.1). We next measure the impact of this rate-limit on the TPR by lowering it. We cannot simulate this behavior accurately by "down-sampling" our data as we have aggregated the flows to remove the source port and IP addresses. Instead, we change the rate-limit threshold across all US regions for two months and collect a new dataset. We cannot do a complete sensitivity analysis with multiple rate-limits as this would be costly – we need to capture at least a month's worth of data for training. Therefore, we limit our experiment to just one threshold: 900 flows per 10 second interval. Such a rate limit will result in data loss for over 30% of our training samples (Figure 8). The results show a *significant* decrease in accuracy (Figure 17-a): 80% TPR for 40% FPR. We conclude PrivateEye needs to capture as many flows as possible to maintain high accuracy.

More training data helps compensate for lower rate-limits (Figure 17-b). We start with 92% TPR for 59.6% FPR, and by just adding 8 more days worth of data to the training set it can reach the same TPR for 23.23% FPR.

PrivateEye is unable to detect the type of compromise. PrivateEye cannot specify the type of malware installed on the VM. Our dataset contains additional information about a *fraction* of the compromised VMs e.g., compromised through SSH or RDP brute-force, malware found, port-scanner, port-sweeper, SQL-injection, RDP/SSH brute-forcer, spammer, or others. However, PrivateEye has low accuracy when identifying the type of compromise (70% TPR for 1% FPR) – it is known that multi-class classifiers tend to have lower accuracy [42]. Besides, not all detections have this additional information.

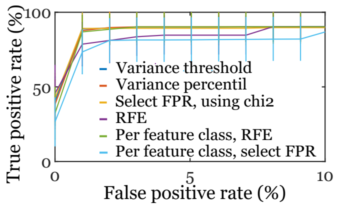


Figure 15: ROC for different feature selection methods.

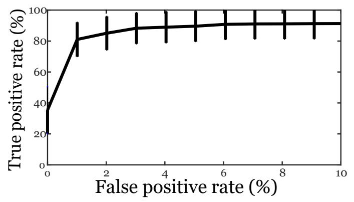


Figure 16: ROC: Impact of the choice of CDFs.

8.3 Performance overhead

One motivation for designing PrivateEye was the need to scale intrusion detection systems to the entire DC. Here we evaluate whether PrivateEye meets this scalability requirement:

- What is the memory and CPU usage of the CA?
- What is the impact of the CA on ongoing traffic?
- What is the expected load on the AA?
- What is the overhead of training the AA?

The CA has low CPU and memory overhead. The CA is deployed across *all* production hosts of a large cloud provider for over 2 years and includes data for over 15,000,000 VMs and 300,000 VNets in the US alone. We chose 100 hosts randomly from DCs across the globe and recorded the CA's CPU and disk usage both in the morning and afternoon. Figure 18 shows its memory usage. Its CPU usage remained bellow 0.1% across all hosts at all times.

The CA does not impact ongoing traffic. Our experience with the CA is that it causes no impact on ongoing traffic. This is in part because the CA has lower priority when obtaining the lock on the vSwitch table. We instrumented the CA on one host to record the time spent, from *user space*, in each query to the vSwitch. The time captures the time spent in contention on the vSwitch table's read-lock and the time it takes to read x entries (where x is the CA rate-limit). There are 8 VMs on the host. We run a SYN flood attack against one of them to simulate different levels of load. We show the time for attacked and non-attacked VMs (Table 3). The results show both the rate-limit and the load (volume of traffic) on a given VM affect the time spent querying the vSwitch for data about that VM but not for data about other VMs. The CPU usage of the CA remained below 0.1% throughout this experiment.

The load on the AA is acceptable. PrivateEye's AA is trained offline using data from the 5% of VMs monitored by the OBDs. The trained model is distributed across each DC region to serve detections every 10 minutes. To quantify the load the AA will have to handle, we looked at the number of flow records per second the CA captured in three regions

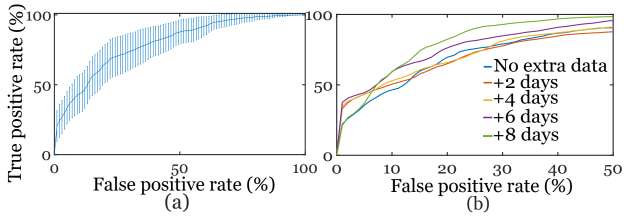


Figure 17: (a) ROC with a rate-limit of 900 flows per 10 seconds. (b) ROC when more data is added.

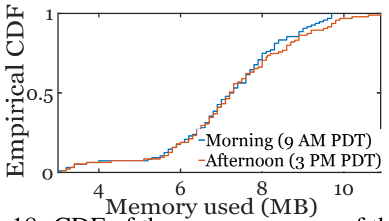


Figure 18: CDF of the memory usage of the CA.

both in the morning and afternoon (Figure 19). The highest rate is around 900,000 flows per second. We expect this volume of data can be easily processed in 10 minute periods.

Training overhead is low. We use Bayesian Optimization to configure our RFs which resulted in an ensemble of 158 trees with a maximum depth of 4. The average training time for such an RF is 5 minutes and $56s \pm 3.76s$ with our two month training set and using 172.87 GB of RAM (peak). The high memory usage is because of the SKlearn implementation which holds the entire dataset in memory.

9 Discussion

This section presents a discussion of the challenges and limitations of a detection system like PrivateEye.

PrivateEye's accuracy. PrivateEye is only as accurate as the OBDs it uses. Thus, our evaluation focuses on comparing PrivateEye to OBDs. OBDs tend to be highly accurate as they are the only systems protecting 1^{st}-party VMs. But, it is challenging to *guarantee* a VM is legitimate. In reality, a legitimate VM is one that passes all detectors deployed in the DC. Should some of these VMs, in fact, be compromised it may cause PrivateEye to have mispredictions (§8). Similarly, we do not know precisely when a VM was compromised but only when it was detected and some of our compromised data may be from when the VM was not yet compromised. Our results in §8 show PrivateEye to be resilient to mislabels.

Generalizing PrivateEye to the entire DC. In our evaluations we partitioned the set of labeled VNets to create a train/test set to emulate how PrivateEye will be used in practice. The VMs we tested PrivateEye on were those which were absent from the training set. These VMs run both Windows and Linux and span a variety of workloads including those of customers who have subscribed to OBDs. We are reasonably confident PrivateEye can detect most compromised VMs. We would have liked to show a small-scale evaluation of PrivateEye where we manually investigated VMs that are not protected by OBDs. However, we were not able to obtain permission to do so.

rate-limit	SYN flood flows per second	Query time (μs) for non attacked VMs	Query time (μs) for attacked VM
900	0	585.3 ± 34.1	-
5000	0	2707.27 ± 105.4	-
10000	0	5097.56 ± 261.0	-
900	10000	780.9 ± 110.4	75276.6 ± 6387.5
5000	10000	2933.3 ± 251.7	71961.0 ± 15607.0
10000	10000	5690.1 ± 430.2	75115.5 ± 18360.8
900	50000	504.6 ± 29.2	70760.4 ± 4611.2
5000	50000	2713.4 ± 272.4	75180.8 ± 1639.3
10000	50000	5699.0 ± 289.4	46922.6 ± 9214.63

Table 3: Profiling the impact on vSwitch read-lock. The times are mean across all samples collected over a 1 minute interval.

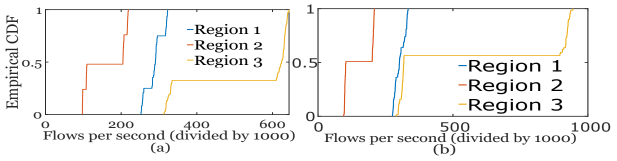


Figure 19: Flows/s captured by the CA. (a) Morning (9 AM-11AM UDT) (b) Afternoon (7PM - 9 PM UDT).

Need for retraining. We can retrain PrivateEye to adapt to changes in malware behavior. Retraining may not be enough to allow PrivateEye to detect all such changes, but the change in malware should increase the time to compromise of VMs due to the attackers needing to avoid conspicuous network flows.

The use of ML. Our work on PrivateEye is the first privacy preserving compromise detection system that can run at scale. Other, for example graph theoretic approaches, could be used as well. It is unclear how such algorithms can adapt to changes in malware behavior. This is clearer for ML models where the re-training of the model can update the system. Graph-based NNs are also applicable [59, 60]. It may be possible to improve the accuracy of PrivateEye even further by using these models. This is a subject of future work.

Deploying the AA. We are currently in the process of deploying the AA using Resource Central [61]. Resource Central allows us to store our model in Azure and serves predictions using that model at run-time. It is highly scalable, and we have already used it to deploy several other ML models in production. However, there are still other questions that we still need to answer, for example, who should build the CDFs and what CDFs are best?

Attacks against PrivateEye. PrivateEye itself may be targeted by attackers to reduce the operator's detection capabilities. Adversarial learning [62] is a sub-field of machine learning that studies such attacks. A study of how to guard against such attacks is beyond the scope of this work.

Higher number of 3^{rd}-party compromises. 87% of our compromised data were 3^{rd}-party VMs, however, the majority of monitored VMs in our data are 1^{st}-party VMs. The higher number of 3^{rd}-party compromises is likely due to the tighter protections on 1^{st}-party VMs. We looked at how this could influence our results and conducted preliminary experiments where we eliminated all 1^{st}-party VMs from the data. On average we achieved 86.71% TPR for 3% FPR (83% FPR for 2.5% TPR). We expect accuracy to improve by increasing the number of samples (the dataset has far fewer datapoints than the original) and by re-tuning the model.

Prior work. We have extensively evaluated the performance of PrivateEye and have also compared it to systems currently deployed across the provider's production DCs. Most prior work are not comparable to PrivateEye as they require packet captures (or introspection) we cannot collect because we need customer permission. PrivateEye is not a replacement for these systems but is designed for DC operators (as opposed to customers). Customers can continue using alternative solutions to protect their VMs. In §7.1 we compared the CA to NetFlow and showed it captures more information. The aggregations and anonymizations applied by the CA prohibit direct comparisons of our AA to NetFlow-based approaches.

Ethical considerations. We annonymized all privacy sensitive information during data collection and *removed them* after feature construction. We conducted all experiments using data from a large cloud provider. The data was either collected from 1^{st}-party VMs under the operator's control or from 3^{rd}-party VMs where the operator had customer permission to monitor the VM. We had explicitly asked permission for deploying the honeypots described in §2 and monitored them closely to ensure they did not cause harm to other VMs. These VMs were not co-located with other VMs.

10 Related Work

We discussed a number of prior works in §1,§7.1, and §7.2. Most do not provide the scalability and privacy characteristics we need [14–20].

Two lines of previous work relate to PrivateEye. One shows the multitude of today's security problems and challenges providers face [6, 63–65]. The other identify malware, compromises, and other types of bad behavior [66–75]. These works can further be divided into two categories:

Network traffic-based Compromise Detection. PrivateEye does not focus on a specific type of attack – it detects any compromise the OBDs can detect. Many prior works identify specific types of bad behavior [5, 5, 9, 12, 19, 21, 22, 47, 57, 76–98]. Some focus on anomaly detection ([99] is a survey of such approaches). Nemean [100] builds intrusion signatures from honeypot packet traces. SNARE finds spammers using packet headers [21]. The work in [18] uses event ordering to identify malware families. VMWall constructs application-aware firewalls aimed at stopping attacks [11]. [90] uses domain knowledge about worms to construct informative features, thus avoiding using IPs (our features capture most of the same information). These works focus on a specific attack which prevents them from comprehensive protection of VMs. Works such as [14–16] rely on packet captures or DPI [101] to identify malicious flows. Packet captures and DPI at DC-scale across all hosts are not possible due to the prohibitive performance overhead. The work of [102] relies on malware propagation to detect the source of attack through analyzing network traffic at key vantage points. However, it does not target identifying the infected nodes. The work of [83] encodes IP addresses through per-source entropies to detect worm attacks; such an approach removes most of the informative properties of individual destination IPs. Such an approach is typically useful when detecting worms and volumetric attacks. The work of [103, 104] discuss other limitations of this approach. Perhaps the closest work to ours is [105] which uses IPFix data from core routers to detect machines that are compromised through SSH brute force attacks. Aside from targeting a specific form of compromise, [105] is based on a fixed set of rules derived through observing a limited set of malware. It is difficult for the approach to adapt to changes in malware behavior. Finally, [47] uses external IP reputation sources to reduce its false positives. This violates our privacy requirements. In addition, we have observed the intersection of malicious IPs reported by commercial IP reputation services and IPs attacking our VMs to be relatively small ($< 10\%$).

Many such works [9, 19, 21, 22, 57] are trained using labels from commercial anti-virus software. Our approach enables us to build a detector that is customized to the cloud because our OBDs detect malicious behavior that occurs in real cloud VMs that are running real workloads. Using OBDs deployed on production VMs running real workloads for labeling allows PrivateEye to avoid problems faced by works such as [106–108] which run malware in emulation mode or in a sandbox to obtain signatures for detection. Many malware can detect when in emulation mode and therefore change their behavior in such situations [18].

Binary-based Compromise Detection. One approach collects malware binaries from honeypots, constructs features from them and then uses Support Vector Machines [7]. Another, clusters binaries found on compromised machines based on their structure, runtime behavior, and the context of the host [8]. Netbait [9] crowd-sources probes gathered from (distributed) infected machines to detect worms. Another approach analyzes memory dumps to construct signatures of the in-memory behavior of malware [10]. Unlike these approaches, PrivateEye performs its classification using networking data alone. Today's privacy requirements, performance constraints, and the new mandates from GDPR make the use of binary and memory inspection techniques in DCs difficult.

Other works also exist [90, 109–113]. Many of these inspired PrivateEye, however, in contrast to these works, PrivateEye's design is aimed at running at scale, having strong privacy requirements, and compliance with GDPR mandates.

11 Conclusion

PrivateEye is a privacy preserving compromise detection system that runs at DC-scale without requiring customer permission. It achieves a true positive rate of 95.77% for a 1% false positive rate.

12 Acknowledgements

The authors would like to thank the anonymous reviewers and our shepherd Hamed Haddadi for their constructive feedback. The authors would also like to thank: Srikanth Kandula, Omid AlipourFard, Vincent Liu, Ricardo Bianchini, Weidong Cui, Ryan Becket, Mina Tahmasbi, Akshay Narayan, Srinivas Narayana, Shadi Noghabi, Robert MacDavid, Ishai Menache, Landon Cox, Hongqiang Liu, and Yibo Zhu for their feedback on our early manuscripts.

References

[1] Microsoft-Inc. Azure security center. https://azure.microsoft.com/en-us/services/security-center/.

[2] Google-Inc. Stackdriver logging. https://cloud.google.com/logging/.

[3] Amazon security solutions. https://aws.amazon.com/mp/scenarios/security/malware/.

[4] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 129–143. ACM, 2016.

[5] Rui Miao, Rahul Potharaju, Minlan Yu, and Navendu Jain. The Dark Menace: Characterizing Network-based Attacks in the Cloud. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, pages 169–182, 2015.

[6] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, 2009.

[7] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. *Learning and Classification of Malware Behavior*, pages 108–125. 2008.

[8] Ahmet Salih Buyukkayhan, Alina Oprea, Zhou Li, and William Robertson. *Lens on the Endpoint: Hunting for Malicious Software Through Endpoint Data Analysis*. Springer International Publishing, 2017.

[9] Brent N Chun, Jason Lee, Hakim Weatherspoon, and Brent N Chun. Netbait: a distributed worm detection service. *Intel Research Berkeley Technical Report IRB-TR-03*, 33, 2003.

[10] Antonio Bianchi, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Blacksheep: Detecting Compromised Hosts in Homogeneous Crowds. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 341–352.

[11] Abhinav Srivastava and Jonathon Giffin. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 39–58, 2008.

[12] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. Jackstraws: Picking command and control connections from bot traffic. In *USENIX Security Symposium*, volume 2011. San Francisco, CA, USA, 2011.

[13] Hao Zhang, Danfeng Daphne Yao, and Naren Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 39–50. ACM, 2014.

[14] Dmitri Bekerman, Bracha Shapira, Lior Rokach, and Ariel Bar. Unknown malware detection using network traffic classification. In *Communications and Network Security (CNS), 2015 IEEE Conference on*, pages 134–142. IEEE, 2015.

[15] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[16] Michael R Watson, Angelos K Marnerides, Andreas Mauthe, David Hutchison, et al. Malware detection in cloud computing infrastructures. *IEEE Transactions on Dependable and Secure Computing*, 13(2):192–205, 2016.

[17] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20(1):343–357, 2016.

[18] Aziz Mohaisen, Andrew G West, Allison Mankin, and Omar Alrawi. Chatter: Classifying malware families using system event ordering. In *Communications and Network Security (CNS), 2014 IEEE Conference on*, pages 283–291. IEEE, 2014.

[19] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. 2008.

[20] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[21] Shuang Hao, Nadeem Ahmed Syed, Nick Feamster, Alexander G. Gray, and Sven Krasser. Detecting Spammers with SNARE: Spatio-temporal Network-level Automatic Reputation Engine. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[22] Guofei Gu, Roberto Perdisci, Junjie Zhang, Wenke Lee, et al. BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet

Detection. In *USENIX Security Symposium*, pages 139–154, 2008.

[23] Zainab Abaid, Mohsen Rezvani, and Sanjay Jha. Malwaremonitor: an sdn-based framework for securing large networks. In *Proceedings of the 2014 CoNEXT on Student Workshop*, pages 40–42. ACM, 2014.

[24] General data protection regulation. `https://ec.europa.eu/info/law/law-topic/data-protection_en`.

[25] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[26] The good, the bad, and the ugly of ml for networked systems. `https://www.microsoft.com/en-us/research/video/the-good-the-bad-and-the-ugly-of-ml-for-networked-systems/`.

[27] Daniel Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 315–328, 2017.

[28] Xin Li, Fang Bian, Mark Crovella, Christophe Diot, Ramesh Govindan, Gianluca Iannaccone, and Anukool Lakhina. Detection and identification of network anomalies using sketch subspaces. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 147–152. ACM, 2006.

[29] David Wittman. List of commonly used passwords. `https://github.com/DavidWittman/wpxmlrpcbrute/blob/master/wordlists/1000-most-common-passwords.txt`, 2015.

[30] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 479–491. ACM, 2015.

[31] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[32] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast string searching on pisa. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 21–28. ACM, 2019.

[33] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM workshop on Virtual machine security*, pages 11–22. ACM, 2009.

[34] Windows defender. `https://www.microsoft.com/en-us/windows/windows-defender/`.

[35] Clam av. `https://www.clamav.net/`.

[36] Kun-Lun Li, Hou-Kuan Huang, Sheng-Feng Tian, and Wei Xu. Improving one-class svm for anomaly detection. In *Machine Learning and Cybernetics, 2003 International Conference on*, volume 5, pages 3077–3081. IEEE, 2003.

[37] WJRM Priyadarshana and Georgy Sofronov. Multiple break-points detection in array cgh data via the cross-entropy method. *IEEE/ACM transactions on computational biology and bioinformatics*, 12(2):487–498, 2015.

[38] David S Matteson and Nicholas A James. A nonparametric approach for multiple change point analysis of multivariate data. *Journal of the American Statistical Association*, 109(505):334–345, 2014.

[39] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[40] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[41] Geoffrey E Hinton, Alex Krizhevsky, and Sida D Wang. Transforming auto-encoders. In *International Conference on Artificial Neural Networks*, pages 44–51. Springer, 2011.

[42] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 440–453.

[43] John T Kent. Information gain and a general measure of correlation. *Biometrika*, 70(1):163–173, 1983.

[44] Gilles Louppe and Manoj Kumar. Bayesian otimization with skopt. `https://scikit-optimize.github.io/notebooks/bayesian-optimization.html`, 2016.

[45] Jianping Hua, Zixiang Xiong, James Lowey, Edward Suh, and Edward R Dougherty. Optimal number of features as a function of sample size for various classification rules. *Bioinformatics*, 21(8):1509–1515, 2004.

[46] Chris Fleizach, Michael Liljenstam, Per Johansson, Geoffrey M Voelker, and Andras Mehes. Can you infect me now?: malware propagation in mobile phone networks. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 61–68. ACM, 2007.

[47] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 129–138. ACM, 2012.

[48] Keith Winstein and Hari Balakrishnan. Tcp ex machina: computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 123–134. ACM, 2013.

[49] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the 2017 ACM SIGCOMM Conference*, pages 197–210.

[50] Hans-Peter Deutsch. Principle component analysis. In *Derivatives and Internal Models*, pages 539–547. Springer, 2002.

[51] Benoit Claise. Cisco systems netflow services export version 9. 2004.

[52] Benoit Claise. Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information. 2008.

[53] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, and Hui Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 165–176. ACM, 2006.

[54] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98. ACM, 2017.

[55] Chao Chen, Andy Liaw, Leo Breiman, et al. Using random forest to learn imbalanced data. *University of California, Berkeley*, 110:1–12, 2004.

[56] Area under the curve. `http://gim.unmc.edu/dxtests/roc3.htm`.

[57] Greg Cusack, Oliver Michel, and Eric Keller. Machine learning-based detection of ransomware using sdn. In *Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 1–6. ACM, 2018.

[58] Feature selection techniques in sklearn. `http://scikit-learn.org/stable/modules/feature_selection.html`.

[59] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[60] Xavier Bresson and Thomas Laurent. An experimental study of neural networks for variable graphs. 2018.

[61] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2017.

[62] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[63] Danny Yuxing Huang, Hitesh Dharmdasani, Sarah Meiklejohn, Vacha Dave, Chris Grier, Damon McCoy, Stefan Savage, Nicholas Weaver, Alex C Snoeren, and Kirill Levchenko. Botcoin: Monetizing stolen cycles. In *NDSS*. Citeseer, 2014.

[64] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security and Privacy Magazine*, Vol. 8(No. 6):pp. 24–31, 2010.

[65] John P John, Alexander Moshchuk, Steven D Gribble, Arvind Krishnamurthy, et al. Studying spamming botnets using botlab. In *NSDI*, volume 9, pages 291–306, 2009.

[66] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, volume 4, pages 4–4, 2004.

[67] Archit Gupta, Pavan Kuppili, Aditya Akella, and Paul Barford. An empirical study of malware evolution. In *Communication Systems and Networks and Workshops, 2009. COMSNETS 2009. First International*, pages 1–10. IEEE, 2009.

[68] Ting-Kai Huang, Bruno Ribeiro, Harsha V Madhyastha, and Michalis Faloutsos. The socio-monetary incentives of online social network malware campaigns. In *Proceedings of the second ACM conference on Online social networks*, pages 259–270. ACM, 2014.

[69] John P John, Fang Yu, Yinglian Xie, Arvind Krishnamurthy, and Martín Abadi. Heat-seeking honeypots: design and experience. In *Proceedings of the 20th international conference on World wide web*, pages 207–216. ACM, 2011.

[70] Paul Barford and Vinod Yegneswaran. An inside look at botnets. In *Malware detection*, pages 171–191. Springer, 2007.

[71] Vinod Yegneswaran, Paul Barford, and Vern Paxson. Using honeynets for internet situational awareness. In *Proceedings of the Fourth Workshop on Hot Topics in Networks (HotNets IV)*, pages 17–22. Citeseer, 2005.

[72] Vinod Yegneswaran, Paul Barford, and Johannes Ullrich. Internet intrusions: Global characteristics and prevalence. 2002.

[73] Jianxing Chen, Romain Fontugne, Akira Kato, and Kensuke Fukuda. Clustering spam campaigns with fuzzy hashing. In *Proceedings of the AINTEC 2014 on Asian Internet Engineering Conference*, page 66. ACM, 2014.

[74] Seth Hardy, Masashi Crete-Nishihata, Katharine Kleemola, Adam Senft, Byron Sonne, Greg Wiseman, Phillipa Gill, and Ronald J Deibert. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *USENIX Security Symposium*, pages 527–541, 2014.

[75] Luca Invernizzi, Stanislav Miskovic, Ruben Torres, Christopher Kruegel, Sabyasachi Saha, Giovanni Vigna, Sung-Ju Lee, and Marco Mellia. Nazca: Detecting malware distribution in large-scale networks. In *NDSS*, volume 14, pages 23–26, 2014.

[76] Holly Esquivel, Aditya Akella, and Tatsuya Mori. On the effectiveness of ip reputation for spam filtering. In *Communication Systems and Networks (COMSNETS), 2010 Second International Conference on*, pages 1–10. IEEE, 2010.

[77] Christoph Dietzel, Anja Feldmann, and Thomas King. Blackholing at ixps: On the effectiveness of ddos mitigation in the wild. In *International Conference on Passive and Active Network Measurement*, pages 319–332. Springer, 2016.

[78] Alefiya Hussain, John Heidemann, and Christos Papadopoulos. A framework for classifying denial of service attacks. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 99–110. ACM, 2003.

[79] Urbashi Mitra, Antonio Ortega, John Heidemann, and Christos Papadopoulos. Detecting and identifying malware: A new signal processing goal. *IEEE Signal Processing Magazine*, 23(5):107–111, 2006.

[80] Calvin Ardi and John Heidemann. Leveraging controlled information sharing for botnet activity detection. In *Proceedings of the 2018 Workshop on Traffic Measurements for Cybersecurity*, pages 14–20. ACM, 2018.

[81] Katerina Argyraki and David R Cheriton. Scalable network-layer defense against internet bandwidth-flooding attacks. *IEEE/ACM Transactions on Networking (ToN)*, 17(4):1284–1297, 2009.

[82] KyoungSoo Park, Vivek S Pai, Kang-Won Lee, and Seraphin B Calo. Securing web service by automatic robot detection. In *USENIX Annual Technical Conference, General Track*, pages 255–260, 2006.

[83] Arno Wagner and Bernhard Plattner. Entropy based worm and anomaly detection in fast ip networks. In *Enabling Technologies: Infrastructure for Collaborative Enterprise, 2005. 14th IEEE International Workshops on*, pages 172–177. IEEE, 2005.

[84] Mark Allman, Paul Barford, Balachander Krishnamurthy, and Jia Wang. Tracking the role of adversaries in measuring unwanted traffic. *SRUTI*, 6:6–6, 2006.

[85] Paul Barford and Mike Blodgett. Toward botnet mesocosms. *HotBots*, 7:6–6, 2007.

[86] Theophilus Benson and Balakrishnan Chandrasekaran. Sounding the bell for improving internet (of things) security. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*, pages 77–82. ACM, 2017.

[87] Mobin Javed and Vern Paxson. Detecting stealthy, distributed ssh brute-forcing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 85–96. ACM, 2013.

[88] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX security symposium*, volume 286. San Diego, CA, 2004.

[89] Ayesha Binte Ashfaq, Maria Joseph Robert, Asma Mumtaz, Muhammad Qasim Ali, Ali Sajjad, and Syed Ali Khayam. A comparative evaluation of anomaly detectors under portscan attacks. In *International Workshop on Recent Advances in Intrusion Detection*, pages 351–371. Springer, 2008.

[90] M Zubair Shafiq, Syed Ali Khayam, and Muddassar Farooq. Improving accuracy of immune-inspired malware detectors by using intelligent features. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 119–126. ACM, 2008.

[91] Ayesha Binte Ashfaq, Zainab Abaid, Maliha Ismail, Muhammad Umar Aslam, Affan A Syed, and Syed Ali Khayam. Diagnosing bot infections using bayesian inference. *Journal of Computer Virology and Hacking Techniques*, 14(1):21–38, 2018.

[92] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. Botgrep: Finding p2p bots with structured graph analysis. In *USENIX Security Symposium*, volume 10, pages 95–110, 2010.

[93] José Jair Santanna, Ricardo de O Schmidt, Daphne Tuncer, Joey de Vries, Lisandro Z Granville, and Aiko Pras. Booter blacklist: Unveiling ddos-for-hire websites. In *Network and Service Management (CNSM), 2016 12th International Conference on*, pages 144–152. IEEE, 2016.

[94] Pavlos Lamprakis, Ruggiero Dargenio, David Gugelmann, Vincent Lenders, Markus Happe, and Laurent Vanbever. Unsupervised detection of apt c&c channels using web request graphs. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 366–387. Springer, 2017.

[95] Hongyu Gao, Jun Hu, Christo Wilson, Zhichun Li, Yan Chen, and Ben Y Zhao. Detecting and characterizing social spam campaigns. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 35–47. ACM, 2010.

[96] Vincentius Martin, Qiang Cao, and Theophilus Benson. Fending off iot-hunting attacks at home networks. In *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*, pages 67–72. ACM, 2017.

[97] Zesheng Chen, Chuanyi Ji, and Paul Barford. Spatial-temporal characteristics of internet malicious sources. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 2306–2314. Citeseer, 2008.

[98] Sakil Barbhuiya, Zafeirios Papazachos, Peter Kilpatrick, and Dimitrios S Nikolopoulos. Rads: Real-time anomaly detection system for cloud data centres. *arXiv preprint arXiv:1811.04481*, 2018.

[99] Monowar H Bhuyan, Dhruba Kumar Bhattacharyya, and Jugal K Kalita. Network anomaly detection: methods, systems and tools. *Ieee communications surveys & tutorials*, 16(1):303–336, 2014.

[100] Vinod Yegneswaran, Jonathon T Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantic aware signatures. In *USENIX Security Symposium*, pages 97–112, 2005.

[101] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. *ACM SIGCOMM Computer Communication Review*, 45(4):213–226, 2015.

[102] Vyas Sekar, Yinglian Xie, David Maltz, Michael Reiter, and Hui Zhang. Toward a framework for internet forensic analysis. In *ACM HotNets-III*, 2004.

[103] George Nychis, Vyas Sekar, David G Andersen, Hyong Kim, and Hui Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 151–156. ACM, 2008.

[104] Mobin Javed, Ayesha Binte Ashfaq, M Zubair Shafiq, and Syed Ali Khayam. On the inefficient use of entropy for anomaly detection. In *RAID*, pages 369–370. Springer, 2009.

[105] Rick Hofstede, Luuk Hendriks, Anna Sperotto, and Aiko Pras. Ssh compromise detection using netflow/ipfix. *ACM SIGCOMM Computer Communication Review*, 44(5):20–26, 2014.

[106] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, volume 4, pages 351–366, 2009.

[107] Thomas Blasing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software (MALWARE 2010)*, pages 55–62. IEEE, 2010.

[108] Jaime Devesa, Igor Santos, Xabier Cantero, Yoseba K Penya, and Pablo García Bringas. Automatic behaviour-based analysis and classification system for malware detection. *ICEIS (2)*, 2:395–399, 2010.

[109] Mainack Mondal, Bimal Viswanath, Allen Clement, Peter Druschel, Krishna P Gummadi, Alan Mislove, and Ansley Post. Defending against large-scale crawls in online social networks. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 325–336. ACM, 2012.

[110] Facebook to contact 87 million users affected by data breach. `https://www.theguardian.com//technology//2018//apr//08//facebook-to-contact-the-87-million-users-affected-by-data-breach`.

[111] Bimal Viswanath, Muhammad Ahmad Bashir, Mark Crovella, Saikat Guha, Krishna P Gummadi, Balachander Krishnamurthy, and Alan Mislove. Towards detecting anomalous user behavior in online social networks. In *USENIX Security Symposium*, pages 223–238, 2014.

[112] Stefan Frei, Thomas Duebendorfer, and Bernhard Plattner. Firefox (in) security update dynamics exposed. *ACM SIGCOMM Computer Communication Review*, 39(1):16–22, 2008.

[113] Grant Ho, Aashish Sharma Mobin Javed, Vern Paxson, and David Wagner. Detecting credential spearphishing attacks in enterprise settings. In *Proceedings of the 26rd USENIX Security Symposium (USENIX Security?17)*, pages 469–485, 2017.

[114] `http://heartbleed.com/`.

A Using PrivateEye

PrivateEye is designed as a preliminary detector and its detections should be followed up with more expensive techniques (e.g., [15]). These techniques are computationally expensive and require customer permission. PrivateEye's role is to reduce the number of VMs that need to be investigated and to protect all VMs at all times with low overhead.

Sometimes, obtaining customer permissions takes too long e.g., if a new vulnerability is discovered that could be exploited by compromised machines (e.g., Heartbleed [114]) the provider may not have time to obtain permission. The operator may choose to move suspect VMs to a sandbox[4] until the appropriate patch is applied to all VMs and devices. If obtaining customer permission in time is not possible, PrivateEye can be used to decide whether a VM should be moved or not. But what are the implications of using PrivateEye as the only compromise detection system in the DC?

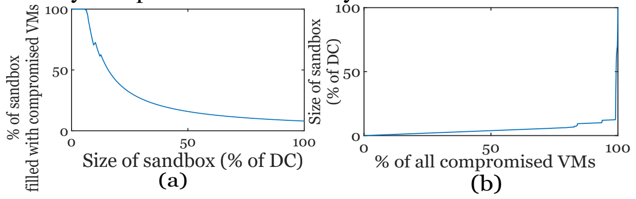


Figure 20: **(a) Fraction of the sandbox occupied by compromised VMs. (b) Sandbox size needed to isolate $x\%$ of compromised VMs.**

For a sandbox size k the operator needs to decide which VMs to put in the sandbox. Using PrivateEye's scores, one choice is to move the top k most suspicious VMs. Figure 20-a illustrates what fraction of the sandbox would be occupied by compromised VMs for different values of k. As the sandbox size increases the utility of the sandbox diminishes–an increasing number of legitimate VMs end up in the sandbox. The choice of k is a tradeoff between the number of VMs that need to be migrated and the number of compromised VMs captured. Figure 20-b examines this tradeoff in our dataset. The operator needs to consider the combination of these graphs when choosing an appropriate k. The larger the sandbox, the more effective it is in reducing the number of compromised VMs outside the sandbox. However, larger k means that it is more likely to place legitimate VMs in the sandbox impacting their performance. To avoid penalizing legitimate VMs, the operator can choose not to migrate a VM if its score is below a threshold. Finding the optimal threshold depends on the operator's needs.

[4]A sandbox could be a host that only runs suspect VMs (limit damage of side-channels) or where more stringent ACLs are imposed.

Telekine: Secure Computing with Cloud GPUs

Tyler Hunt
The University of Texas at Austin

Zhipeng Jia
The University of Texas at Austin

Vance Miller
The University of Texas at Austin

Ariel Szekely
The University of Texas at Austin

Yige Hu
The University of Texas at Austin

Christopher J. Rossbach
The University of Texas at Austin and VMware Research

Emmett Witchel
The University of Texas at Austin

Abstract

GPUs have become ubiquitous in the cloud due to the dramatic performance gains they enable in domains such as machine learning and computer vision. However, offloading GPU computation to the cloud requires placing enormous trust in providers and administrators. Recent proposals for GPU trusted execution environments (TEEs) are promising but fail to address very real side-channel concerns. To illustrate the severity of the problem, we demonstrate a novel attack that enables an attacker to correctly classify images from ImageNet [17] by observing only the timing of GPU kernel execution, rather than the images themselves.

Telekine enables applications to use GPU acceleration in the cloud securely, based on a novel GPU stream abstraction that ensures execution and interaction through untrusted components are independent of any secret data. Given a GPU with support for a TEE, Telekine employs a novel variant of API remoting to partition application-level software into components to ensure secret-dependent behaviors occur only on trusted components. Telekine can securely train modern image recognition models on MXNet [10] with 10%–22% performance penalty relative to an insecure baseline with a locally attached GPU. It runs graph algorithms using Galois [75] on one and two GPUs with 18%–41% overhead.

1 Introduction

GPUs have become popular computational accelerators in public clouds. Accuracy improvements enabled by GPU-accelerated computation are driving the success of machine learning and computer vision in application domains such as medicine [38, 88] transportation [67], finance [32], insurance [69], gaming [89], and communication [70].

Unfortunately, it is currently impossible to run GPU workloads in the cloud without trusting the provider, eliminating cloud GPUs as an option for security-conscious users. Users must trust the provider because the provider controls the layer of privileged software responsible for management and provisioning. Even dedicated cloud instances (e.g., Amazon's EC2 dedicated hosts [1]) run the provider's virtualization software, making GPUs vulnerable to malicious or curious

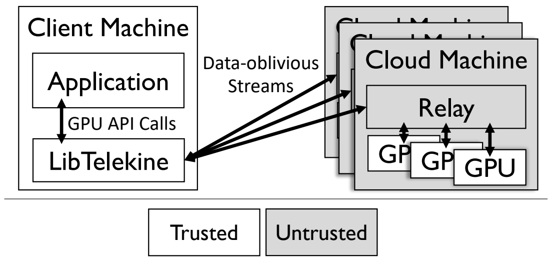


Figure 1. Telekine components and their organization.

cloud administrators. Virtualization software runs at a host machine's highest privilege level, exposing a wide attack surface that includes GPU memory, execution context, and firmware. Finally, unfettered visibility into host/device communication exposes both data and timing channels.

Trusted execution environments (TEEs) should, in principle, make the cloud an option for users who refuse to trust the provider. TEEs provide a hardware root of trust, allowing users to access cloud compute resources without trusting provider software–including the privileged software of the hypervisor and operating system. TEE hardware protects the privacy and integrity of user code and data from administrators and from attackers who control privileged software. TEEs exist currently on Intel CPUs via software guard extensions (SGX) [44], ARM CPUs via TrustZone [59], and RISC-V CPUs via Keystone [55]. Researchers have proposed GPU-based TEEs [100] and TEE extensions for GPUs [45], though none have been built or deployed. However, as we argue below, a design that simply composes components that run in hardware-supported CPU and GPU TEEs will fail to provide strong security due to side channels.

GPU-accelerated applications have three main software components: (1) an API and a user library (e.g., CUDA [66] or HIP [39]) that provides high-level programming functionality and executes on a CPU; (2) CPU-side control code at the user and the system level that manages communication with the GPU, and (3) GPU kernels (programs) that execute on the GPU device itself. It is the data and code that moves between the CPU and GPU that potentially creates side channels visible to CPU-side code.

An attacker can extract meaningful information from the execution time of code on the GPU, which privileged software can compute on the CPU by observing communication with the GPU. For example, we demonstrate a novel attack on image recognition machine learning models that allows malicious system software to correctly classify images from ImageNet [17] used as input to the model. By observing only the timing of a model trained to classify images (the image model), we build a new model (the timing model) that can classify images based on the execution timing of layers in the image model. Even if a security-conscious user encrypts their input data (and decrypts it on the GPU), a cloud provider's system administrator can use the timing information of GPU kernels (measured on the CPU) in the image model to classify its input images. We train the timing model to distinguish images of two classes with 78% accuracy. For more classes, accuracy decreases but stays above random guessing.

We propose Telekine, a system that enables the secure use of cloud GPUs without trusting the platform provider. GPU TEEs provide a secure execution environment but leave the user open to side channels when communication depends on secret data. Telekine makes communication with the GPU TEE data oblivious, that is, completely independent of secrets contained in the input data. Data obliviousness is a strong property that excludes the existence of side-channel attacks against CPU-side code and host/device communication whose observable behavior (e.g., timing, memory accesses, DMA sizes, etc.) depends on secret input data.

Telekine has three components (shown in Figure 1): libTelekine that runs on a trusted user machine (a *client*), GPUs physically attached to a cloud machine (a *server*) that supports GPU TEEs with specific security requirements (§3.1), and the relay which facilitates communication between libTelekine and the GPU. Telekine uses a GPU TEE because it needs a mechanism to protect GPU computation from the cloud provider; a GPU TEE is tailored to that task.

Telekine protects the application and GPU runtime by moving it from the cloud to the client. The advantage of this approach is that the user must already trust their client machine, and the application and user libraries are large and complex and therefore prone to side-channel attacks, making them difficult to secure if they execute in the cloud. The disadvantage is that GPU libraries assume a local GPU with a fast, high-bandwidth connection to the CPU. Telekine decouples the user library from low-level GPU control by interposing on the GPU API and efficiently forwarding these calls to the server (a technique known as API remoting) which has been used to virtualize GPUs [6, 8, 22, 23, 30, 33, 50, 57, 58, 85, 101, 107], but to our knowledge has never been used for security. A client using Telekine does not need to have a GPU installed.

Telekine treats the CPU-side control code on the cloud server ("Relay" in Figure 1) as completely untrusted, almost as if it were part of the network. The client machine establishes a cryptographically secure channel directly with the code executing on the cloud GPU. The network and the CPU-based code on the server can delay the computation, but cannot compromise its privacy or integrity.

Telekine secures the communication between the client machine and the cloud GPU by transforming the user's GPU API calls into *data-oblivious streams*. Data-oblivious streams are similar to constant time defenses [3] in that they aim to remove timing channels by ensuring that observable events are deterministic regardless of secrets. Telekine constructs data-oblivious streams by reducing all API calls to a sequence of code execution (`launchKernel`) and data movement (`memcpy`) commands. It then schedules these commands at a fixed rate, possibly creating new commands, or splitting `memcpy` commands into fixed-size pieces. Fixed-sized, fixed-rate communication is data oblivious; it ensures that any observable patterns are independent of the input data and therefore devoid of side-channel information. Fixed-rate communication is not a novel way to eliminate side channels, but Telekine's design shows how to apply it efficiently to modern GPU-based computing.

Given that Telekine requires a GPU TEE, it is logical to wonder why it does not use a CPU TEE. After all, putting the application and programming libraries into a CPU TEE would reduce the latency and increases the bandwidth for communication between libTelekine and the GPU. Unfortunately, Intel and ARM TEEs do not prevent side channels as part of their threat model [48, 78]. Keystone [55] and Komodo [25] intend to address side channels for RISC-V and ARM respectively, but work is ongoing. Also, making existing applications data oblivious is difficult for programmers, requires access to source code (not needed by Telekine), and often slows down a program greatly (e.g., Opaque [109] slows down data analytics by 1.6–46×). Should future CPU TEEs evolve to address side channels, Telekine can use them. Much of Telekine focuses on securing the communication between trusted components, which can be an improved CPU TEE and a GPU TEE or they can be the client machine and server GPU TEE, as they are in our prototype.

Telekine is the first system to offer efficient, secure execution of GPU-accelerated applications on cloud machines under a strong and realistic threat model. We use Telekine to secure several GPU-accelerated applications via two frameworks: the MXNet [10] machine learning framework and the Galois graph processing system [75]. On a realistic testbed Telekine provides strong secrecy and integrity guarantees, including side-channel protection. MXNet [10] training for three different, modern image recognition models incur a 10–22% performance penalty relative to a baseline with a locally attached GPU. MXNet inference for the same models over a connection from Austin, TX to the Vultur's Dallas, TX datacenter [102] incurs a penalties of 0-8% for batch sizes of 64 images. Telekine runs graph algorithms using Galois [75] on one and two GPUs with 18%–41% overhead.

This paper makes the following contributions.

- We demonstrate a CPU-side timing attack on deep neural networks that allows a compromised OS to correctly classify images in encrypted input (§4).
- We provide a design and prototype for Telekine, a system that eliminates CPU-based side-channel attacks against a GPU TEE with a novel variant of API remoting to execute secret-dependent code on the GPU TEE and a trusted client (§5).
- We thoroughly evaluate the performance, robustness, and security of Telekine protecting a variety of important workloads on one and two GPUs: machine learning and graph processing (§7).

2 Threat model

In all current cloud GPU platforms, the cloud provider's privileged software, and hence administrators, can gain easy access to GPU state, creating a significant attack surface including explicit channels such as GPU memory, firmware, and execution context. Work in this area agrees on the vulnerability of GPU state to privileged software [45, 100].

Telekine assumes a powerful adversary who controls all software on the platform, including privileged software such as device drivers, the host operating system and hypervisor. This captures typical cloud platforms, where the platform provider has full control over all software, and attackers can run malicious code on the same physical device as a target cloud application [81]. A malicious provider, a malicious administrator, or an OS-level attacker can use their control of privileged software to steal the secrets of tenants. We assume that the adversary cannot, however, compromise hardware–the physical GPU package.

Telekine assumes a GPU TEE, with capabilities similar to current research proposals like Graviton [100]. The details can vary, but a GPU TEE establishes secure memory on the GPU device and provides a protocol to initiate a computation that can be remotely attested to start from the correct state (code and initial data) and execute privately and without interference from the CPU side. We provide additional detail on Telekine's TEE requirements in Section 3.1.

GPU TEEs do not, by themselves, secure communication with the CPU and our attack (§4) shows how much information there is in the precise timing of CPU/GPU communication. Telekine protects communication with the GPU, guaranteeing that the adversary cannot learn about input data directly or through side channels, including timing channels.

While secure control of a GPU has been proposed [45, 100], there has been little work securing side channels. These side channels undercut the security of the TEE. In addition to the timing attack we developed (§4), AES key extraction using shared GPU hardware [31, 46, 47] has been demonstrated. And recent side-channel attacks [64] have shown practical methods to fingerprint websites using performance counters observed during GPU rendering in the browser.

2.1 Guarantees

Telekine provides the following secrecy properties which prevent any explicit or implicit data flow from input data to an external observer.

S1 (content): Messages are encrypted to ensure their content cannot be directly read by an observer.

S2 (timing): The transmit schedule for messages is fixed. Any transmission delays are independent of input data.

S3 (size): The size of each message is fixed. Telekine pads and/or splits messages to achieve fixed-sized messages.

Telekine also provides the following integrity properties to ensure that any result the user receives is either a result that could have been generated by a GPU hosted by a completely benign cloud provider, or an error.

I1 (content): The content of all communication is protected by an end-to-end integrity check; a message authentication code (MAC) allows Telekine to detect modifications, returning an error if any are detected.

I2 (order): Each message carries a sequence number which allows Telekine to detect out of order messages. The sequence numbers also prevent replay attacks.

I3 (API-preserving): Commands issued by the application should affect GPU state in the same way they would on a local GPU, regardless of any transformations that Telekine applies.

GPU commands have semantics that Telekine must maintain for correctness. For example, GPU runtimes expose a *stream* [71] abstraction to application code. API calls issued by the application on the same stream are executed serially in the order they were issued. A kernel launched from a particular stream will block the completion of subsequent API calls on that stream until that kernel terminates. Applications can have many streams which map to different command queues exposed by hardware. API calls made on separate streams can be executed in parallel. Telekine must respect the data dependence semantics of streams.

2.2 Limitations.

Physical side channels and denial of service attacks are out of scope. In situations where an adversary monitoring physical side channels like temperature [62], power [54], or acoustical emanations [11] is a concern, Telekine would need to be augmented with other techniques to maintain security. In our threat model, a cloud provider wishing to deny service can always do so, e.g., by interrupting the network or refusing to run user processes.

Telekine provides clients a mechanism to disguise their end-to-end runtime but does not impose policy. Applications can choose the most efficient policy for their security needs. We believe end-to-end runtime is a poor predictor of input data (and our experiments in Section 4 bear this out), further justifying the clients setting policy.

3 GPU background

Applications use GPUs through high-level, vendor-provided APIs such as CUDA [66] and HIP [39]; they include a user-level runtime and OS-level driver that communicate through a combination of `ioctl` system calls and memory-mapped command queues. The driver is responsible for creating mappings from virtual memory to physical MMIO regions. After these privileged operations are complete, any software that has a mapping (user or OS) may communicate directly with the device using registers or command queues exposed through the MMIO regions.

While memory management, synchronization, and other features (e.g., IPC, power management, etc.) require interaction with the driver state (e.g., creating and managing memory mappings), a workload that pre-allocates all of its required GPU memory and uses only data transfer and kernel launch primitives can function completely by writing commands into the GPU's command queue. It is possible to construct and submit these commands without referring to any state maintained by either the runtime or the driver. As we show in Section 5, this property enables Telekine's relay to be effectively stateless.

3.1 GPU TEE

Telekine requires a TEE on the GPU and Graviton [100] is a detailed proposal from the literature that provides the basic functionality that any GPU TEE (or indeed any TEE) should provide: secrecy for GPU code and input data, integrity for the GPU computation, and remote attestation for the computation's initial state. Graviton achieves most of its functionality by changing the GPU firmware, so it does not require extensive changes to the GPU hardware itself (neither does Telekine). This is achievable because the modern GPU firmware runs on a fully programmable control processor [68]. We explain GPU TEE functionality by saying what the GPU does, but the implementation could be firmware, hardware, or both.

The integrity, secrecy, and ordering of the commands sent to the GPU are ensured by a secure channel. Before computation begins, the client machine and the GPU agree on a shared symmetric key via a key exchange protocol (e.g., Diffie-Hellman). The client uses this key to send commands using a protocol like transport layer security (TLS) which provides a secure channel ([100] §5.2).

The integrity of the computation is assured by the GPU, which checks the initial execution conditions and attests these conditions to the remote user, who can verify that the expected code has been loaded into the expected address range with the expected permissions, and that the hardware generating the attestation is genuine. There are many variations on remote attestation, but it is a common feature for modern enclaves like SGX [14] and Keystone [55]. Telekine expects the GPU to have been initialized with all of the GPU kernels the application intends to launch and any initial data when attestation has completed.

Telekine and Graviton split GPU memory into untrusted and trusted regions so the untrusted host OS/Hypervisor can DMA into untrusted GPU memory, enabling efficient data transfers; the GPU can then copy data between untrusted GPU memory and trusted memory. This mechanism provides GPU memory protection even though the IOMMU is under control of the untrusted kernel. Telekine and Graviton disable unified memory, which allows privileged CPU code to demand page GPU memory and exposes side-channel memory access information.

The GPU TEE should turn off or refuse to report the state of any performance counters. Recent GPU side-channel attacks [28, 64] have successfully used timing data from GPU performance counters.

Due to Telekine's focus on side channels, it has requirements beyond the previously proposed GPU TEEs. These requirements are more straightforward to provide than the core TEE functionality.

Eliminate GPU side channels. Some TEE designs allow different tenants/principals to execute concurrently (e.g., SGX, Keystone), sharing the underlying hardware. Concurrent execution is attractive from a utilization perspective but it provides a rich side-channel attack surface which has plagued the security of CPU TEE designs. Telekine assumes side channels from concurrent principals (e.g., memory access timing and bandwidth) do not exist on the GPU TEE. A conservative design which prevents hardware side channels is to disallow concurrent execution. Graviton TEEs scrub their state (e.g., registers, memory, caches) after resources are freed so there is no danger of tenants observing transient state from previous computation.

Conceal kernel completions. GPUs signal the CPU via an interrupt when a kernel has completed its execution. Interrupt timing leaks information about the kernel's runtime. Rather than rely on interrupts, Telekine uses data-oblivious streams (§5.1) that include tagged buffers that allow the GPU to communicate computational results back to the client. The platform only sees DMA from the GPU to untrusted CPU memory at a fixed rate.

Support no-op kernel launches. Dependences between GPU kernels often cause the launch of one kernel to wait for another's completion, which provides indirect timing information. The GPU TEE must support a no-op kernel launch command so that Telekine can generate cover traffic ensure the adversary sees kernel launches at a fixed rate.

Timely command consumption. The GPU TEE should consume its command queue independently of how long kernels take to execute on the GPU. If the GPU waits until each kernel completes before dequeueing the next launch command, it can fall behind the input queue fill rate, allowing the input queue to fill. The adversary can detect this situation by observing how often the encrypted queue content changes, creating a proxy for kernel execution time. The GPU should consume command queue entries at a fixed rate, discard the no-ops, and store the real commands internally until they can be executed. Telekine can hold back real kernel launches and send no-op launches in their places to ensure these internal GPU queues do not fill up.

3.2 Communicating with GPUs

GPUs can be connected to the CPU memory interconnect (integrated) or to the PCIe bus (discrete). We focus on PCIe-attached GPUs because they are preferred in performance-focused settings like the cloud due to their higher memory bandwidth and better performance.

The PCIe interfaces provide two forms of communication: memory-mapped I/O (MMIO) and direct memory access (DMA). MMIO re-purposes regions of the physical memory address space for device communication. Contiguous physical ranges, or *BARs* (base-address-regions) are reserved by the hardware, and the hardware redirects loads/stores targeting those regions to the device. Modern GPUs use MMIO BARs to expose registers for configuring the device, and frequently accessed device memory (e.g., command queues).

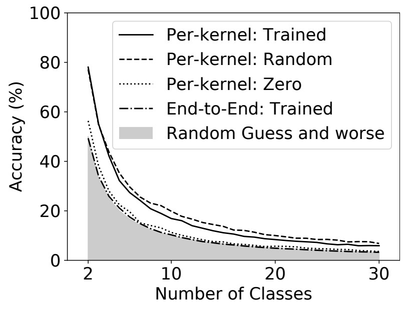
Any software that can obtain a mapping to MMIO can potentially communicate with the GPU to control it (through a register or command queue interface) or read/write its memory (through an MMIO memory BAR or by configuring DMA transfers to/from it). Telekine assumes GPU TEE support similar to Graviton [100] to prevent MMIO access to GPU status and configuration registers during secure execution.

The hypervisor and/or host operating system controls the PCIe bus, which routes packets to multiple devices connected to the PCIe root complex in a tree topology. Packets in transit to/from the GPU may be visible to other devices. Privileged host software may change the routing topology dynamically and can install pseudo-devices that allow it to sniff traffic. Securing communication with the GPU must defend against these passive and active PCIe attacks.

4 Example side-channel attack

Telekine addresses software attacks launched by an adverary resident on a cloud host, such as those launched by a malicious system administrator or a network-based attacker who has compromised the platform's privileged software. These attacks use privileged software to compromise the privacy or integrity of user code and data. Telekine is particularly focused on protecting against timing channels because effective, general-purpose attacks using timing channels have recently been demonstrated at the architecture level [53, 60, 84, 96], the OS level [97, 105], and the GPU programming level [46, 47]. Modern CPU TEEs exclude side channels from their threat model [31, 48, 78], leaving current hardware-supported security primitives vulnerable to side-channel attack. Telekine offers a unique and efficient security solution for cloud resident, GPU-based computation.

We demonstrate a proof-of-concept attack on machine learning inference in which the adversary uses the execution timing of individual GPU kernels to learn information about encrypted input data. Our attack allows privileged software on the cloud



(a) Batches of size 1

Purity	Accuracy
0.25	29.3%
0.40	33.0%
0.60	41.0%
0.80	50.0%
0.90	56.1%
1.00	65.4%

(b) Batches of size 32

Figure 2. Accuracy of multiclass classification for side-channel attacks. (a) shows the accuracy for a baches of size 1 with an increasing number of classes. (b) shows the accuracy for batches of size 32, 4 classes, and varies how much of the batch contains the target image (purity)

host to correctly classify images using only the timing of GPU kernel execution obtained on the CPU. The attacker can train their timing model on their own input, they do not need the victim's training data. The image data remains encrypted while on the CPU and the attack does not require any access to GPU architectural or microarchitectural state (including GPU timers).

Attack basics. Convolutional neural networks (CNNs) are a popular neural network architecture for analyzing images [37, 40, 91]. Each network consists of multiple layers, including convolutions, which are good at detecting features of the input image that the remainder of the network can use to classify the image. When CNNs are executed on a GPU, the computation for each layer roughly corresponds to the execution of a single GPU kernel. While the actual mapping between layers and kernels is often more complex, the intuition behind our attack is that the timing of the execution of certain CNN layers (and hence their GPU kernels) indicates the presence or absence of certain features within the input image. This makes the per-layer execution time itself a rich feature.

Telekine defeats the attack by removing the adversary's ability to infer the timing of individual kernels. The adversary retains only the ability to measure the end-to-end runtime of the inference task. However, our data shows that end-to-end runtime provides very little predictive value, making the attack not much more accurate than randomly guessing (Figure 2a). Telekine gives users the mechanism to disguise their end-to-end execution time, should they decide to do so (§2.2).

Attack details. We demonstrate this attack on ResNet50 [37], a CNN widely used for image recognition, using the timing of GPU kernel completion events as detected by the operating system on the CPU (though we monitor a function in the GPU's user-level runtime for ease of implementation). We evaluate the accuracy of our attack using 5-fold cross validation.

We start with a pre-trained model for the standard ImageNet [17] dataset which contains 1,000 different image classes. Figure 2a shows the accuracy of distinguishing image classes based on the timing of the pre-trained model's layers (Per-kernel: Trained), versus the same attack using only end-to-end timing information (End-to-end: Trained). The accuracy of the per-kernel classifier is startlingly good for small numbers of classes: 78% for two classes, 55% for three and 42% for four. As the number of classes of input images increases, the accuracy of our classification declines, but it remains much better than random guessing, outperforming guessing by over 1.9× even among 30 input image classes.

We believe the root cause of the attack is timing dependent GPU operations, probably multiplication by zero. We compare a pre-trained model (Per-kernel: Trained with no zero-valued weights), a randomly initialized model (Per-kernel: Random with 0.2% zero-valued weights), and a model whose weights are all zero (Per-kernel: Zero with 100% zero-valued weights). The zero model has bad accuracy that is close to random guessing. A randomly initialized model is best, followed by the pre-trained model.

These results were generated using MXNet [10] ported to HIP on the ROCm version 1.8 stack for AMD GPUs which is used in the prototype; we saw similar results on the 2.9 version. Preliminary tests showed that this specific attack is much less powerful on NVIDIA GPUs.

Batched classification. Because inference is often done in batches, we examine the accuracy of a batched attack. We construct batches by splitting each ImageNet class into disjoint training and test sets. Images are then randomly sampled from each of these sets to form the batches.

We present the accuracy of our attack when distinguishing four ImageNet classes in batches of size 32 (Figure 2b.) Each batch is made up of the given fraction of images from a primary class (Purity), and randomly selected images from the remaining three classes. Our objective is to correctly identify the primary class.

Batches help, with the accuracy of our attack improving with larger batch sizes. Larger batches execute more operations, effectively amplifying the timing signal our attack relies on. Moreover, larger batches smooth out execution timings for outlier images which would otherwise be less recognizable to our attack model. When distinguishing four classes (Figure 2b), the batched attack is better than random guessing even when only 25% of the input images come from the target class. The accuracy increases with higher batch purity, outperforming single images by up to 64%.

5 Design

Telekine secures GPU-based computation from active attackers, including side-channel threats. Side channels include the execution timing of individual GPU kernels as well as data movement to and from the GPU. Telekine achieves its security by transforming an application's computation so that all communication—including data movement—among trusted components is data oblivious. Telekine only trusts the client machine and the in-cloud GPU TEE and must, therefore,

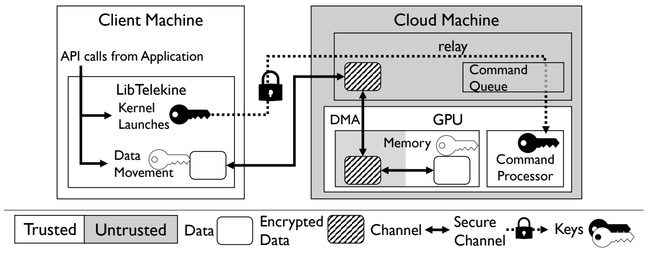


Figure 3. Detailed Telekine overview.

efficiently coordinate the computation between these entities, even though communication occurs over a wide area network, rather than over higher-bandwidth, lower-latency fabric like a data center network or a PCIe bus.

Telekine consists of three components (depicted in Figure 1 with detail in Figure 3).

- LibTelekine: a library that intercepts GPU API calls from the application and transparently transforms the calls into a data-oblivious command stream.
- Relay: an untrusted process that runs in the cloud and directs the client's command stream to the GPU.
- GPU: a GPU (or multiple GPUs) with TEE support that meets Telekine's requirements (see §3.1 for details).

LibTelekine is linked into the application running on the client. During its execution, the application issues a stream of GPU commands through the normal GPU API. Similar to normal API remoting [8, 21, 101], libTelekine redirects API calls made by the client to a server process with a GPU runtime–the relay on the cloud machine. Telekine treats the relay almost as if it were part of the network, relying on it to communicate with the GPU but protecting that communication with end-to-end techniques. The relay is not part of Telekine's trusted computing base.

All communication between libTelekine and the GPU is protected with authenticated encryption (AES-GCM [24] in our prototype) and sequence numbers. This creates a secure channel satisfying the secrecy property *S1 (content)* and the integrity properties *I1 (content)* and *I2 (order)* (described in §2.1), ensuring that the GPU commands issued by libTelekine can only be read by the GPU, and any tampering or reordering is detectable. However, by observing when messages are exchanged with the GPU (regardless of whether they are encrypted), the adversary can get timing information about the computation on the GPU.

Telekine's goal is to remove all timing information from the encrypted stream of GPU commands. It removes timing information by sending commands (GPU runtime API calls like `launchKernel` and `memcpy`) at a fixed rate, independent of input data. Fixed rating is a simple idea, but Telekine must overcome two major challenges to fix-rate GPU communication.

1. Different GPU command types are distinguishable because they have different sizes and they result in different communication patterns with the GPU. (e.g.,

launchKernel commands interact with MMIO ring buffers and memcpy commands are handled using DMA). Telekine must ensure that the attacker's ability to distinguish between these commands conveys no information about the input data.

2. Conventional GPU command streams (§2.1) exhibit a variety of data-dependent behavior whose timing is externally visible (e.g., a kernel launch after a data transfer will wait for the data transfer to finish). Telekine must maintain the ordering semantics induced by such data dependencies.

Telekine introduces a new primitive to overcome these challenges: *data-oblivious streams*. Data-oblivious streams transparently replace conventional GPU streams (and applications may have more than one), maintaining their semantics while making their communication with the GPU data oblivious. First, they separate commands by type, and schedule each type independently. Second, they split, pad, and batch commands of each type so that the encrypted payload is always the same size for messages of that type, satisfying *S3 (size)*. Third, they inject management commands as needed to maintain data-dependencies across message types, satisfying *I3 (API-preserving)*. Finally, data-oblivious streams send the transformed commands according to a fixed schedule, satisfying *S2 (timing)*.

The relay, privileged software on the cloud machine, and the network stack can delay commands since they are under complete control of the (possibly adversarial) cloud provider. However, they cannot delay commands in a way that leaks input data because all observable behavior of the trusted computing base (including its timing) is independent of input data.

5.1 Data-oblivious stream construction

Constructing data-oblivious streams only requires reasoning about memcpy and launchKernel commands. The TEE takes care of initialization (§3.1). The only other runtime commands deal with stream synchronization, and Telekine transforms those commands into memcpy and launchKernel commands as well (discussed fully in §5.4). memcpy commands are visible to the untrusted host's privileged software because GPU drivers use DMA for efficient data transfers. In Telekine, the data itself is protected and copied to/from a fixed staging area in untrusted GPU memory so the destination/-source of the memcpy does not leak information.

Conventional GPU streams can create timing channels from memcpy and launchKernel commands because a memcpy command waits for all previous launchKernel commands on the same stream. To eliminate this channel, Telekine uses two GPU streams to construct a single data-oblivious stream. Telekine uses one GPU stream to launch the application's kernels; this stream is called the ExecStream. Telekine uses the other stream—called the XferStream—to move data to and from the GPU. Telekine ensures that commands on the XferStream never leak information about the kernel execution time by waiting for commands on the ExecStream.

The ExecStream. Application kernels are all launched on the ExecStream. LibTelekine maintains a queue of the launchKernel commands requested by the application and releases the commands in order according to the fixed-rate schedule. The GPU consumes these commands independently of any ongoing kernel execution and buffers them internally since their execution must be serialized according to GPU stream semantics. Telekine honors data dependences between memcpy and launchKernel commands by inserting data management kernels that block the progress of the ExecStream by spinning until the data is in place.

The XferStream. Data transfers requested by the application are launched on the XferStream. Unlike launchKernel commands, memcpy commands are directional (i.e., client-to-GPU and GPU-to-client), and directions are detectable. For example, because the adversary can observe interaction with the network, it can differentiate between messages that came over the network in transit to the GPU, and messages copied from the GPU that are being sent over the network. LibTelekine maintains separate queues for each direction and schedules them independently to avoid leaking information. Data for client-to-GPU transfers starts on the client, flows through the relay and into untrusted memory on the GPU. LibTelekine then enqueues a kernel, which moves the data from the untrusted staging memory into trusted GPU memory. Similarly, in the GPU-to-client direction, Telekine first enqueues a launchKernel on the XferStream to move the data into untrusted GPU memory, then issues a memcpy to copy it to the relay where it can be transferred over the network back to the client.

Fixed-size commands. Telekine ensures that all memcpy commands are the same size by splitting and padding the memcpy commands issued by the application to a standard size. When there are no pending memcpy commands, Telekine maintains the same rate of data flow by scheduling dummy, standard-sized memcpys to/from a staging buffer. Similarly, all launchKernel commands are padded to the same size (320 bytes in our prototype). When no launchKernel command is available, Telekine schedules no-op launchKernel commands.

Schedules. Any schedule Telekine uses for GPU communication is secure so long as it does not depend on the data being protected. Our prototype uses simple schedules which send a fixed number of fixed-sized commands after each fixed-time interval. For instance, Telekine might launch 16 kernels on the ExecStream every 3 milliseconds, and send then receive 4MB of data every 6 milliseconds on the XferStream.

Schedules can leak the category. While scheduling work at a fixed rate is a well-known technique to avoid side-channel leakage, the exact schedule is relevant to performance. We

Algorithm 1 Telekine's replacement functions for `memcpy` and `launchKernel`. Splitting and padding steps are omitted for brevity.

```
 1: function LaunchKernel($kern$,$args$...)
 2:     Enqueue($kernelQueue$,$\{kern,args\}$)
 3: end function
 4:
 5: function MemcpyH2D($src$,$dst$)
 6:     $buf$ ←ChooseTaggedBuffer()
 7:     LaunchKernel(copy_in, $buf$, $dst$)
 8:     Enqueue($dataQueueH2D$, $\{src, buf\}$)
 9: end function
10:
11: function MemcpyD2H($src$, $dst$)
12:     $buf$ ←ChooseTaggedBuffer()
13:     LaunchKernel(copy_out, $src$, $buf$)
14:     Enqueue($dataQueueD2H$, $\{buf, dst\}$)
15: end function
```

Algorithm 2 Periodic tasks performed by Telekine according to the schedule. Encryption and decryption steps are omitted for brevity.

```
 1: loop                                   ▷ ExecStream Thread
 2:     if Empty($kernelQueue$) then
 3:         $op$ ←no_op
 4:     else
 5:         $op$ ←Dequeue($kernelQueue$)
 6:     end if
 7:     WaitForScheduledTime()
 8:     RemoteLaunchKernel($op$)
 9: end loop
10:
11: loop              ▷ XferStream Client-to-GPU (H2D) Thread
12:     if Empty($DataQueueH2D$) then
13:         $src$ ←dummy_CPU
14:         $dst$ ←ChooseTaggedBuffer()
15:     else
16:         $\{src, dst\}$ ←Dequeue($dataQueueH2D$)
17:     end if
18:     WaitForScheduledTime()
19:     RemoteMemcpy($src$, $dst$)
20: end loop
21:
22: loop              ▷ XferStream GPU-to-Client (D2H) Thread
23:     if Empty($DataQueueD2H$) then
24:         $src$ ←ChooseTaggedBuffer()
25:         $dst$ ←dummy_CPU
26:     else
27:         $\{src, dst\}$←Peek($dataQueueD2H$)
28:     end if
29:     WaitForScheduledTime()
30:     RemoteMemcpy($src$, $dst$)
31:     if $dst \neq$ dummy_CPU then
32:         if TagMatches($dst$) then
33:             Dequeue($dataQueueD2H$)
34:         end if
35:     end if
36: end loop
```

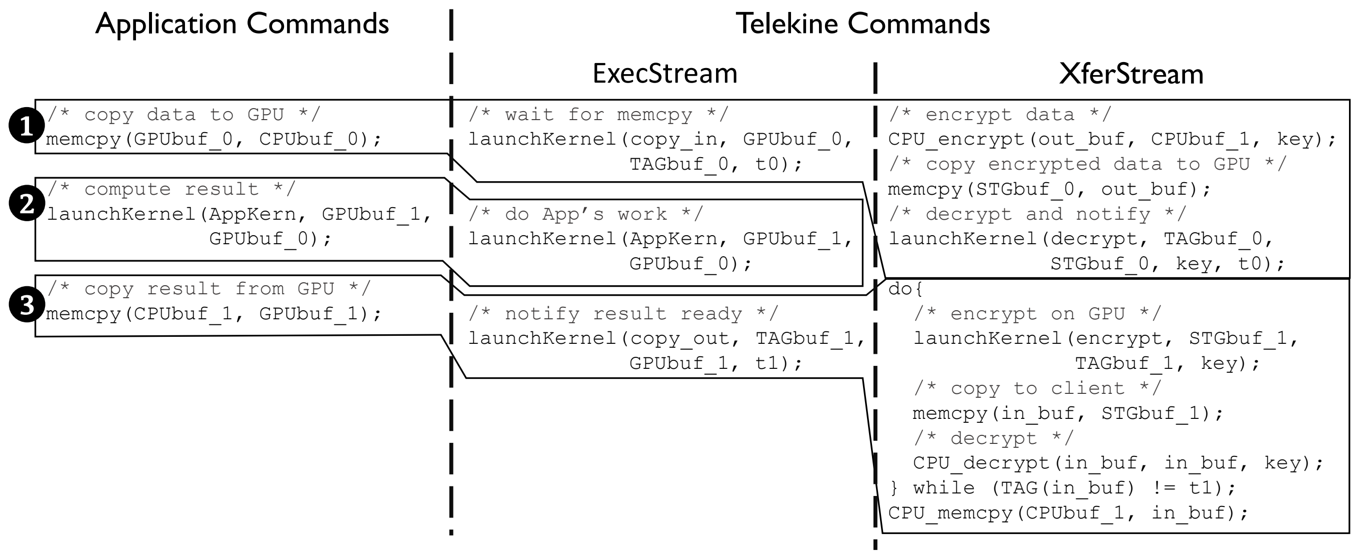
report our schedules in Table 1, and they are the same for all tasks of a given category, e.g., training different machine learning models with MXNet. However, they can differ across categories, e.g., Galois has a different ExecStream schedule from MXNet (§7). Under our threat model, the adversary would be able to differentiate these workloads from their network traffic. A user can always choose a more generic, but lower performing schedule if this is a concern.

5.2 Telekine operation

Algorithm 1 and Algorithm 2 provide a high-level description of Telekine's data-oblivious streams. In Algorithm 1, Telekine intercepts the application's calls to `launchKernel` and `memcpy` and transforms them into interactions with queues: kernelQueue, dataQueueH2D, and dataQueueD2H (splitting, padding, and encryption steps are omitted for brevity). The Telekine threads shown in Algorithm 2 dequeue the commands and release them to the GPU according to the schedule. Telekine waits at lines 7, 18, and 29 for the next available time slot ensuring that interactions with the queues do not influence the timing of messages.

Most `memcpy` commands have strict ordering requirements with respect to kernels that operate on their data. The `memcpy` then `launchKernel` idiom ensures that the launched kernel has fresh data to process. While Telekine decouples `memcpy` commands by scheduling them on their own stream for security, it needs to preserve the original ordering semantics expected by the application. Telekine maintains these semantics by injecting its own data management kernels into the ExecStream (shown on lines 7 and 13 of Algorithm 1) to enforce the ordering expected by the application. These data management kernels operate on *tagged buffers* which Telekine uses to synchronize data access.

Tagged buffers. Tagged buffers are pre-allocated staging buffers on the GPU, each with an associated tag slot. Telekine assigns every `memcpy` operation a tagged buffer and a unique tag, represented by "ChooseTaggedBuffer" in Algorithm 1 and Algorithm 2. Data management kernels producing data (e.g., copying out the result of a kernel computation) write the tag into the tag slot of the chosen tagged buffer after the operation has completed and a memory barrier completes. Data management kernels that consume data (e.g., waiting for data a kernel expects to use as input) wait until the tag slot of the assigned buffer contains the expected value since they cannot be sure the buffer data is valid until the tag value matches its expectation.



```
/* copy data to GPU */
memcpy(GPUbuf_0, CPUbuf_0);
```

```
/* wait for memcpy */
launchKernel(copy_in, GPUbuf_0,
             TAGbuf_0, t0);
```

```
/* encrypt data */
CPU_encrypt(out_buf, CPUbuf_1, key);
/* copy encrypted data to GPU */
memcpy(STGbuf_0, out_buf);
/* decrypt and notify */
launchKernel(decrypt, TAGbuf_0,
             STGbuf_0, key, t0);
```

```
/* compute result */
launchKernel(AppKern, GPUbuf_1,
             GPUbuf_0);
```

```
/* do App's work */
launchKernel(AppKern, GPUbuf_1,
             GPUbuf_0);
```

```
/* copy result from GPU */
memcpy(CPUbuf_1, GPUbuf_1);
```

```
/* notify result ready */
launchKernel(copy_out, TAGbuf_1,
             GPUbuf_1, t1);
```

```
do{
  /* encrypt on GPU */
  launchKernel(encrypt, STGbuf_1,
               TAGbuf_1, key);
  /* copy to client */
  memcpy(in_buf, STGbuf_1);
  /* decrypt */
  CPU_decrypt(in_buf, in_buf, key);
} while (TAG(in_buf) != t1);
CPU_memcpy(CPUbuf_1, in_buf);
```

Figure 4. API calls made by the application and their mapping to underlying commands performed by Telekine.

Data management kernels. Telekine inserts its own data management kernels into the ExecStream which either produce or consume tagged buffers depending on the direction of the transfer. There are two kernels: `copy_in` and `copy_out`. Both kernels take an application-defined memory location, a tagged buffer, and a tag as arguments. For CPU-to-GPU `memcpys`, libTelekine inserts a `copy_in` launch into the ExecStream. The `copy_in` will repeatedly check the tag slot of the buffer, completing the copy to the application's buffer only after verifying the tag slot matches the tag it was given as an argument. To service GPU-to-CPU `memcpys`, Telekine inserts a `copy_out` into the ExecStream after the application kernel which generates the data. The `copy_out` writes the data to the assigned tagged buffer, followed by the tag to signal to Telekine that the data is ready. Since libTelekine runs on the client it has no way of knowing when the copy out has completed until the tagged buffer has been copied back, so it will retry the same GPU-to-CPU copy until the tag is correct corresponding to a complete copy. This is represented by the PEEK operation on line 27 of Algorithm 2, the operation is only dequeued after libTelekine verifies that the `copy_out` kernel did its work on line 32.

GPU-to-GPU data copies. Emerging hardware supports dedicated, high-bandwidth, cross-GPU communication links such as NVLink [26]. NVLink improves cross-GPU data copy efficiency but does not change the fundamental communication mechanisms used in a GPU stack. Telekine currently implements GPU-to-GPU copies as two copies: one from the first GPU back to the client and the second from the client to the second GPU. Direct GPU-to-GPU copies using NVLink would be far more efficient, but to be data oblivious they would have to occur at a fixed rate. We leave this task for future work.

Discussion. The XferStream is carefully constructed so that it never synchronizes with the ExecStream. The XferStream contains DMA operations which the OS can detect; if application kernels on the ExecStream occupy the GPU causing the encryption kernels on the XferStream—and transitively the DMAs—to wait, then the platform can learn some information about kernel execution times. There may still be leakage between the XferStream and the ExecStream because we cannot guarantee that kernels of the former will not interfere with the latter. However, we believe this leakage to be hard to exploit in practice, we have not seen it in any of our benchmarks, and we expect that future GPU features like strict priority [72] or preemption [92] will allow Telekine to seal the leak.

5.3 Data movement example.

Figure 4 shows an example of how Telekine transforms application commands into equivalent, data-oblivious commands on the ExecStream and XferStream. The application issues 3 commands: ❶ copy data to the GPU, ❷ launch a kernel to process that data, and ❸ copy the results of the computation out of the GPU back to the CPU.

❶ : The application requests a `memcpy` from `CPUbuf_0` to `GPUbuf_0`. In response, Telekine chooses a tag, `t0`, and tagged buffer, `TAGbuf_0`, for this operation, then enqueues a kernel, `copy_in`, on the ExecStream. The `copy_in` kernel will spin on the GPU, using atomic operations to check the end of `TAGbuf_0` until it sees `t0`, then copy the contents of `TAGbuf_0` into `GPUbuf_0`. On the XferStream, Telekine encrypts the data, then copies the encrypted data to a staging buffer in untrusted GPU memory `STGbuf_0`. Finally, Telekine launches a kernel, `decrypt`, on the XferStream which reads the encrypted data out of untrusted memory and decrypts it into `TAGbuf_0`. After the data is written, the tag `t0` is appended

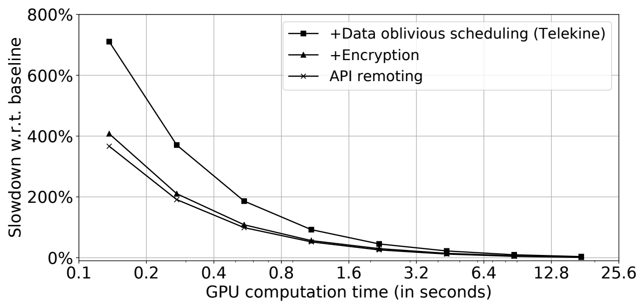


Figure 5. A microbenchmark which shows how Telekine overheads decrease as the running time of the GPU computation increases.

after a memory barrier, signaling to `copy_in` that the data is ready.

❷ : The application launches its kernel, `AppKern`, which processes the data in `GPUbuf_0` and writes its result into `GPUbuf_1`. Since `AppKern` is launched on the ExecStream after `copy_in` it will wait for `copy_in` to complete, ensuring that the data will be in `GPUbuf_0` before `AppKern` starts. The platform cannot detect that `AppKern` has started.

❸ : The application issues a request to copy the results of `AppKern` from `GPUbuf_1` to `CPU_buf1`. In response, Telekine again chooses a tag and tagged buffer, `t1` and `TAGbuf_1` respectively, and immediately enqueues a `copy_out` kernel on the ExecStream. After the application's kernel, `AppKern`, has completed, `copy_out` moves the result of its computation in `GPUbuf_1` into `TAGbuf_1` then atomically appends `t1`. While waiting for `copy_out` to finish, Telekine periodically encrypts `TAGbuf_1` into a staging buffer in untrusted memory, `STGbuf_1`, then issues a `memcpy` operation to copy the contents of `STGbuf_1` to a client-side buffer, `in_buf`. Telekine decrypts `in_buf` and checks the tag. If the tag matches `t1`, `copy_out` and `AppKern` must have completed and the data can be copied into `CPUbuf_1`. If not, this process will be repeated during the next scheduled GPU to client transfer.

5.4 Synchronizing data-oblivious streams

Applications sometimes wish to synchronize with their GPU streams (i.e., wait for all outstanding commands to complete), or synchronize one GPU stream with another (i.e., ensure another stream has completed some operation, n, before this stream starts operation, m). Telekine handles both of these cases by injecting kernels that increment a counter in GPU memory between kernels in the ExecStream. Because of stream semantics, the increment kernel only runs after all previous kernels in the stream, providing an accurate count of how many application kernels have executed. Telekine copies that counter back to the client periodically and can block the application thread until all submitted work has completed.

	ExecStream		XferStream		
Benchmark	Quantum	Size	Quantum	Size	Bandwidth
Microbench	15ms	32kerns	30ms	1MB	533 Mb/s
MXNet	15ms	512kerns	30ms	1MB	533 Mb/s
Galois1	15ms	32kerns	30ms	1MB	533 Mb/s
Galois2	15ms	32kerns	30ms	1MB	533 Mb/s

Table 1. Data-oblivious schedule parameters and the network bandwidth required. MicroBench from §7.1; MXNet from §7.2; Galois1 executes on one GPU, Galois2 on two from §7.3. ExecStream sizes are number of kernel launches, each of which is 320 bytes. XferStream streams contribute twice their size to bandwidth consumption because Telekine copies data in both directions at every quantum.

6 Implementation

The Telekine prototype is based on AMD's ROCm 1.8 [2], an open-source software stack for AMD GPUs. Telekine requires an open-source stack because we split its functionality between user and cloud machines. NVIDIA is generally thought to have higher hardware and software performance as well as better third-party software support. But NVIDIA only officially supports closed-source drivers and runtimes.

LibTelekine and the relay. All applications were ported to use HIP [39], the ROCm CUDA replacement. LibTelekine marshals the arguments of HIP API calls to be sent over a TLS protected TCP connection to the relay to support initialization. The libTelekine and relay prototype are based on code generated by AvA [107]; they total 8,843 and 5,650 lines of C/C++/HIP code respectively (measured by cloc [12]).

GPU TEE. GPU TEE requirements are made explicit in Section 3.1, and most of those requirements are safety properties that do not impact performance. A notable exception is the cryptography required to secure the secrecy and integrity of kernel launch commands. We model the timing of these features by decrypting kernel launch commands in the relay.

7 Evaluation

We quantify the overheads of the security Telekine provides by comparing it to an insecure baseline: applications run on cloud provider machines that offload computation to GPUs directly through the GPU runtime.

We measure Telekine across two testbeds. The first is the *simulated testbed* which simulates wide-area network (WAN) latency and bandwidth, providing a controlled environment for measurement. The second is the *geodist testbed* in which the server and client are geodistributed and connected by the Internet. Both testbeds use the same "cloud machine" (the *server*), which has an Intel i9-9900K CPU with 8 cores @3.60GHz, 32GB of RAM and two Radeon RX VEGA 64 GPUs each with 8GB of RAM. All machines are running Ubuntu 16.04.6 LTS with Linux kernel version 4.13.0, and AMD's ROCm-1.8 runtime and HIP-1.5 compiler.

In the simulated testbed, the client has an Intel Xeon E3-1270 v6 processor with 4 cores @3.8GHz and 32GB of RAM.

	ResNet	InceptionV3	DenseNet
Model size	97.5 MB	90.9 MB	30.4 MB
Input size			
Input image	224x224x3	299x299x3	224x224x3
Batch size	64	64	48
Data size per batch	9.2 MB	16.4 MB	6.9 MB
Single-GPU training baseline			
T-put	20.27 MB/s	11.05 MB/s	13.57 MB/s
T-put (less sync)	22.69 MB/s	11.66 MB/s	17.46 MB/s

Table 2. Overview of machine learning training on MXNet. Input size is given in pixel dimensions, batch size in images per GPU. T-put is throughput.

Both this client and the server are equipped with a Gtek X540 10Gb NIC, which we connect directly. We simulate a client-to-cloud network connection in a controlled environment using `netem` [65], which allows us to add network delays and limit bandwidth. We always limit the bandwidth of the connection to 1Gbps and unless otherwise mentioned we add delays in both directions so that the total round trip time (RTT) is 10ms. These parameters are conservative for a network connection to an edge cloud server [16, 106].

In the geodist testbed, the client is a VM hosted by vultr [102] in their Dallas, TX datacenter (the server is in Austin, TX). The VM has 8 vCPUs and 32GB of RAM. We measured the RTT between the server and this client at 12ms, and the average bandwidth at 877Mbps.

Different applications use different schedules to get good performance, though Table 1 shows strong similarity among the data-oblivious schedules we use for evaluation.

7.1 Telekine performance tradeoff

Figure 5 shows the performance tradeoff for a microbenchmark with 16MB of input and output and a GPU kernel with a configurable running time on the simulated testbed. The different lines show the costs of specific sources of overhead. The "API remoting" line uses the XferStream and the ExecStream over the network. The "+Encryption" line adds encryption to API remoting. Finally, the "Data-oblivious scheduling" line adds the data-oblivious schedule described in Table 1 to encryption. When the GPU kernel executes for only 0.14 seconds, the overhead of Telekine is nearly 8×. Once the computation takes 4.4s the overhead is only 22%. Telekine is a remote execution system; it makes communication more expensive because of its oblivious scheduling as well as network delay and limited bandwidth. It is most efficient when computation dominates communication, which is the case for our benchmarks.

7.2 Machine learning algorithms

We port MXNet [10], a state-of-the-art machine learning library, to run on the HIP runtime. Our port is based on MXNet v1.1.0 (git commit `07a83a03`). We also use AMD's MIOpen library for efficient neural network operators. Some parts of MXNet adaptively choose from different GPU kernel implementations by measuring execution times on the available hardware and choosing the most performant option. To ensure the baseline and Telekine are running the same kernels for measurement purposes, we record the kernels chosen by the baseline, and hard-code those kernel choices for all runs.

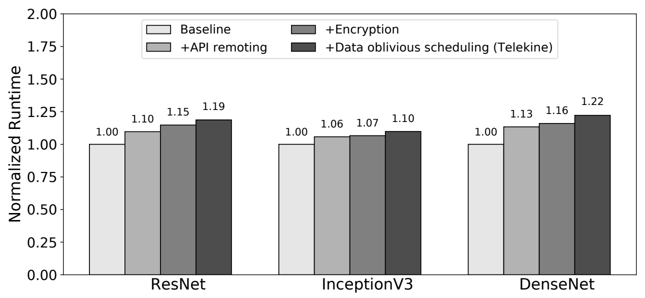


Figure 6. Performance of machine learning training algorithms using a single GPU with Telekine on the simulated testbed.

ResNet	InceptionV3	DenseNet
1.23×	1.08×	1.20×

Table 3. Performance of machine learning training algorithms on Telekine, measured on the geodist testbed.

Optimizing MXNet. We applied several optimizations to MXNet which help to mitigate the fact that Telekine is communicating with the GPU over a WAN:

- The models we evaluate represent the pixel channels of the input bitmaps using 4-byte floating point quantities, even though they range in integer values from 0 to 255. To save network bandwidth, we send bytes instead of floats, reducing bandwidth by 4×. Bytes are changed back floats on the GPU.
- We determined that MXNet was overly conservative in its GPU synchronization strategy and were able to reduce the number of synchronizations it performs by removing unnecessary calls to `hipStreamSyncronize` ("less sync" in Table 2). Telekine also optimizes synchronization calls by using tagged buffers (§5.1) to coordinate data transfers.

Machine learning training. We evaluate the training performance of deep neural networks on Telekine using three state-of-the-art convolutional neural network architectures: ResNet [37], InceptionV3 [91], and DenseNet [40]. All models are trained using the ImageNet dataset (a substantial data set consisting of 1.4 million training images). For ResNet, we use the 50-layer variant. For DenseNet, we use the 121-layer variant. We evaluated all networks using batches size of 64. Table 2 summarizes the input sizes that were used to evaluate the three network architectures.

Figure 6 shows the performance of training three neural nets on Telekine using the simulated tesdbed, normalized to the insecure baseline. The bars break down Telekine's overheads and match the descriptions from Section 7.1. Both Telekine and the baseline use a single GPU. Table 3 shows the same

Batch size	ResNet Base	ResNet Telekine	InceptionV3 Base	InceptionV3 Telekine	DenseNet Base	DenseNet Telekine
Simulated testbed						
1	20	273 (13.7x)	29	259 (8.93x)	26	248 (9.54x)
8	42	270 (6.43x)	65	264 (4.06x)	47	241 (5.13x)
64	233	389 (1.67x)	368	559 (1.52x)	246	405 (1.65x)
256	988	1195 (1.21x)	1520	1806 (1.19x)	946	1163 (1.23x)
Geodist testbed						
1	20	200 (10.0x)	31	205 (6.61x)	26	201 (7.73x)
8	69	241 (3.49x)	111	247 (2.23x)	84	209 (2.49x)
64	462	481 (1.04x)	637	685 (1.08x)	484	483 (1.00x)

Table 4. Latencies (in ms) of machine learning inference workloads with the baseline system (Base in the table) and Telekine.

experiment on the geodist testbed; the results are similar to the simulated testbed.

Machine learning inference. We evaluate neural network inference workloads for ResNet, InceptionV3, and DenseNet with Telekine. For inference, latency is the priority for users, but throughput is still a priority for providers. Batching inference can substantially improve throughput by fully utilizing hardware capabilities and amortizing the overheads from other system components [15]. We evaluate the latency of inference with different batch sizes, ranging from 1 to 256. Our baseline is an insecure server with one local GPU, communicating with the over the network. Table 4 shows the inference latency of three neural networks with different batch sizes. The overheads with on the simulated testbed for batches of size of 256 are 21%, 19%, and 23% for ResNet, InceptionV3, and DenseNet, respectively which are slightly improved compared to the overheads we report for training (§7.2), although the training batch size was 64. With a batch size of 64, the overheads on the simulated testbed inflate to 67%, 52%, and 65%. When we move to the geodist testbed, the performance of the baseline suffers more that Telekine; at batches of size 64, the standard deviation of our measurements exceed the differences between the mean Telekine and baseline runs. Clipper [15] uses an adaptive batch size to meet the latency requirement of the application, which Telekine could adopt.

7.3 Graph algorithms

Galois is a framework designed to accelerate parallel applications with irregular data access patterns, such as graph algorithms [75]. We port Galois's GPU computation to use the HIP runtime instead of CUDA and evaluate it on three graph algorithms: breadth-first search (BFS), PageRank, and single source shortest paths (SSSP). All measurements use the USA roads graph dataset [18]. Figure 5 shows the performance of these applications on Telekine with one and two GPUs. The baseline is an unmodified system with local GPU(s). Baseline performance for single GPU applications is: BFS 54.1s, SSSP 74.6s, Pagerank 60.9s; for two GPUs: BFS 36.4s, SSSP 42.8s. For the input distributed with Galois, two GPU Pagerank slows down, so we do not evaluate it.

Application		Normalized runtime
BFS	(1 GPU)	1.18x
SSSP	(1 GPU)	1.21x
Pagerank	(1 GPU)	1.29x
BFS	(2 GPUs)	1.38x
SSSP	(2 GPUs)	1.41x

Table 5. Performance of Galois applications with Telekine.

RTT (ms)	ResNet	InceptionV3	DenseNet
10	1.19x	1.10x	1.22x
20	1.29x	1.13x	1.37x
30	1.44x	1.16x	1.49x
40	1.53x	1.18x	1.66x
50	1.62x	1.30x	2.09x

Table 6. Normalized runtime of machine learning workloads with respect to network round trip time (RTT).

Telekine imposes moderate overheads on single-GPU Galois applications, adding latency to data transfer times. Galois implements each graph algorithm as a single GPU kernel that is iteratively called until the algorithm reaches termination. Multi-GPU applications exchange data between GPUs through the host after each iteration. Telekine imposes higher overheads for multi-GPU workloads because of increased data movement over the network.

7.4 WAN latency sensitivity

Telekine assumes that the client communicates with the server over a WAN. The greater distances crossed by WANs result in longer round trip times (RTTs). The batching of commands that Telekine does for security also makes it resilient to these increased RTTs, especially when the ratio of GPU computation to communication is high. To demonstrate this we increased the RTT between our machines using `netem` [65] and ran the machine learning training benchmarks for different RTTs (Table 6). Overheads increase with RTT. At 30ms which we measured to be the RTT between the client and an Amazon EC2 instance, the overhead for InceptionV3 is still only 16%.

8 Related Work

Enclave-based security. Several recently proposed systems aim to protect applications from an untrusted platform. Haven [7], SCONE [4], and Graphene-SGX [95] provide an environment to support unmodified legacy applications. Ryoan [43] protects user data from untrusted code and an untrusted platform. VC3 [83] and Opaque [109] provide SGX-protected data processing platforms. None of these systems allow for GPU computation and none of them focus on the communication issues that then arise.

Trusted execution environments on GPUs. HIX [45] extends an SGX-like design with duplicate versions of the enclave memory protection hardware to enable MMIO access from code running in an SGX enclave. This enables HIX to guarantee that a single enclave has exclusive access to the MMIO regions exported by a GPU, in principle, defeating a malicious

OS that wants to interpose or create its own mappings to them. While this design provides stronger GPU isolation than current enclaves, it remains vulnerable to side-channel attacks because communication is not data oblivious.

Graviton [100] supports GPU TEEs based on *secure contexts* that use the GPU command processor to protect memory from other concurrently executing contexts. Similar to Telekine, Graviton secures communication using cryptographic techniques. Telekine can adopt many of Graviton's clever mechanisms for its TEE functionality (§3.1), such as restricting access to GPU page tables without trusting the kernel driver. But Graviton does not protect against side channels, which is Telekine's primary mission.

The opportunity to provide stronger security for GPU-accelerated applications using TEEs and oblivious communication has been observed by others [41].

Securing accelerators. SUD emulates a kernel environment in user space to isolate malicious device drivers [9]. Previous work has explored techniques to support trusted I/O paths, leveraging hypervisor support [103, 110] or system management mode [52]. Our work focuses on the secure use of GPUs with untrusted system software and does not rely on support from the software at lower privilege layers. Border Control [74] addresses security challenges for accelerator-based systems but focuses on protecting the system from a malicious accelerator, rather than Telekine which protects CPU and GPU code from an untrusted platform.

GPU security and protection. Studies have analyzed GPU security properties and vulnerabilities [112]. Frigo et al. [28] demonstrate techniques that leverage integrated GPUs to accelerate side-channel attacks from browser codes using JavaScript and WebGL. PixelVault [98] exploits physical isolation between CPUs and GPUs to implement secure storage for keys, though it was shown to be insecure [112]. CUDA Leaks [77] shows techniques to exfiltrate data from the GPU to a malicious user. Attacks that take advantage of GPU memory reuse without re-initialization are a common theme [36, 56, 111]. Several systems have proposed mechanisms that bring the GPU under tighter control of system software, exploring OS support [34, 49, 63, 82], access to OS-managed resources [51, 86, 87], hypervisor support [20, 30, 33, 85, 90, 93, 101] and GPU architectural support for cross-domain protection [5, 13, 76, 79, 99].

Secure machine learning. Ohrimenko et al. describe an SGX-based system for multi-party machine learning on an untrusted platform [73]. Their data-oblivious algorithm for convolutional neural networks explicitly does not support state-of-the-art operations that are data dependent (e.g., max pooling). Telekine can support any data-dependent operations but requires a GPU TEE. Chiron [42] provides a framework for untrusted code to design and train machine learning models in SGX. Telekine does not support untrusted code, but does allow the use of GPUs which Chiron excludes. CQSTR [108] lets a *trusted* platform operator confine untrusted machine learning code so that it can be securely applied to user data. By contrast, Telekine protects user data from an untrusted platform operator. MLcapsule [35] protects service provider secrets (machine learning model) and client data by running machine learning algorithms in an SGX enclave but does not suggest extensions to allow secure GPU acceleration.

Slalom [94] secures training of DNNs using a combination of TEEs and local GPUs. Slalom's guarantees are achieved by partitioning DNN training into linear layers using matrix multiplication, which are offloaded to a GPU, the remaining operators, which execute on the CPU in a TEE such as SGX. Matrix multiplication is verified and turned private using algorithmic techniques [27], which enables secure GPU offload without requiring GPU TEE support.

Recent work [19, 61] demonstrates how to efficiently *apply* neural networks to encrypted data. As far as we know, today there are no practical techniques for *training* deep neural networks on encrypted data.

API remoting. API remoting [6, 22, 23, 50, 57, 58, 80, 104] is an I/O virtualization technique that interposes a high-level user-mode API. API calls are forwarded to a user-level computing framework [85] on a dedicated appliance VM [101], or on a remote server [23, 50]. To our knowledge, Telekine is the first system to use API remoting as a security technique.

OS-level time protection. Recent extensions to seL4 [29] suggest general OS-level techniques that prevent timing-based covert channels by eliminating sharing of hardware resources that can form the basis of covert channels. The techniques do not yet generalize to I/O-attached accelerators.

9 Conclusion

Telekine enables secure GPU acceleration in the cloud. Telekine protects in-cloud computation with a GPU TEE and application/library computation by placing it on a client machine. It secures their communication with a novel GPU stream abstraction that ensures the execution is independent of input data. Telekine allows GPU-accelerated workloads such as training machine learning models to leverage cloud GPUs while providing strong secrecy and integrity guarantees that protect the user from the platform's privileged software and its administrators.

10 Acknowledgements

We would like to thank Işil Dillig for her invaluable feedback on an early version of this paper. We would also like to thank Bryan Parno (our shepherd) and our anonymous reviewers; their constructive feedback made this paper better than it would have been otherwise. We gratefully acknowledge funding from NSF grants CNS-1900457, CNS-1618563, and CNS-1846169.

References

[1] Amazon. EC2 Dedicated Hosts. https://aws.amazon.com/ec2/dedicated-hosts/. (Accessed: February 12, 2020).

[2] AMD. ROCm, a New Era in Open GPU Computing. https://rocm.github.io/index.html. (Accessed: February 12, 2020).

[3] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards Verified, Constant-time Floating Point Operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1369–1382, 2018.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16. USENIX Association, 2016.

[5] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, MICRO'17. IEEE, 2017.

[6] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *2019 IEEE International Conference on Cluster Computing Workshops and Posters*, CLUSTER WORKSHOPS, September 2010.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14. USENIX Association, 2014.

[8] Bitfusion: The Elastic AI Infrastructure for Multi-Cloud. https://bitfusion.io. (Accessed: February 12, 2020).

[9] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10. USENIX Association, 2010.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Acoustic Cryptanalysis. *Journal of Cryptology*, 30, April 2017.

[12] cloc: Count Lines of Code. https://github.com/AlDanial/cloc. (Accessed: February 12, 2020).

[13] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinman. Supporting Address Translation for Accelerator-Centric Architectures. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA. IEEE, 2017.

[14] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. 2016.

[15] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17. USENIX Association, 2017.

[16] Richard Cziva and Dimitrios P Pezaros. On the Latency Benefits of Edge NFV. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS. IEEE, 2017.

[17] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *The Conference on Computer Vision and Pattern Recognition*, CVPR. IEEE, 2009.

[18] DIMACS. 9th DIMACS Implementation Challenge - Shortest Paths. http://users.diag.uniroma1.it/challenge9/download.shtml, 2005. (Accessed: February 12, 2020).

[19] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *International Conference on Machine Learning*, 2016.

[20] Micah Dowty and Jeremy Sugerman. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[21] J. Duato, AJ. Pena, F. Silla, R. Mayo, and E.S. Quintana-Orti. rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters. In *2010 International Conference on High Performance Computing Systems*, HPCS, 2010.

[22] José Duato, Francisco D Igual, Rafael Mayo, Antonio J Peña, Enrique S Quintana-Ortí, and Federico Silla. An efficient implementation of GPU virtualization in high performance clusters. In *European Conference on Parallel Processing*, Euro-Par'09, pages 385–394, Berlin, Heidelberg, 2009. Springer, Springer-Verlag.

[23] Jose Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. Enabling CUDA acceleration within virtual machines using rCUDA. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[24] Morris Dworkin. NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf, 2007. (Accessed: February 12, 2020).

[25] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using Verification to Disentangle Secure-enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 287–305, New York, NY, USA, 2017. ACM.

[26] Denis Foley. Ultra-Performance Pascal GPU and NVLink Interconnect. In *HotChips*, 2016.

[27] Rusins Freivalds. Probabilistic Machines Can Use Less Running Time. In *IFIP Congress*, pages 839–842, 1977.

[28] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *IEEE Symposium on Security and Privacy*, May 2018.

[29] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *European Conference in Computer Systems*, EuroSys, 2019.

[30] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU transparent virtualization component for high performance computing clouds. In *European Conference on Parallel Processing*, pages 379–391. Springer, Springer, 2010.

[31] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, 2017.

[32] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. Accelerating Financial Applications on the GPU. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 127–136, New York, NY, USA, 2013. ACM.

[33] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009.

[34] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 3–3. USENIX Association, 2011.

[35] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. MLCapsule: Guarded Offline Deployment of Machine Learning as a Service. *CoRR*, abs/1808.00590, 2018.

[36] Ari B. Hayes, Lingda Li, Mohammad Hedayati, Jiahuan He, Eddy Z. Zhang, and Kai Shen. GPU Taint Tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 209–220. USENIX Association, 2017.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[38] Nicole Hemsoth. Medical Imaging Drives GPU Accelerated Deep Learning Developments. https://www.nextplatform.com/2017/11/27/medical-imaging-drives-gpu-accelerated-deep-learning-developments/, November 2017. (Accessed: February 12, 2020).

[39] HIP: Convert CUDA to Portable C++ Code. https://github.com/ROCm-Developer-Tools/HIP. (Accessed: February 12, 2020).

[40] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, volume 1, page 3, 2017.

[41] Tyler Hunt, Zhipeng Jia, Vance Miller, Christopher J. Rossbach, and Emmett Witchel. Isolation and Beyond: Challenges for System Security. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 96–104, New York, NY, USA, 2019. ACM.

[42] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving Machine Learning as a Service. *CoRR*, abs/1803.05961, 2018.

[43] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 533–549. USENIX Association, 2016.

[44] Intel(R) Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014. (Accessed: February 12, 2020).

[45] Insu Jang, Adrian Tang, Taehoo Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, 2019.

[46] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a GPU. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[47] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A Novel Side-Channel Timing Attack on GPUs. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, pages 167–172, New York, NY, USA, 2017. ACM.

[48] Simon Johnson. Intel SGX and Side-Channels. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels, March 2017. (Accessed: February 12, 2020).

[49] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, USENIXATC'11, pages 17–30. USENIX Association, 2011.

[50] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, page 341352. ACM, 2012.

[51] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216. USENIX Association, 2014.

[52] Yonggon Kim, Ohmin Kwon, Jinsoo Jang, Seongwook Jin, Hyeongboo Baek, Brent Byunghoon Kang, and Hyunsoo Yoon. On-demand bootstrapping mechanism for isolated cryptographic operations on commodity accelerators. 62, 7 2016.

[53] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *CoRR*, January 2018.

[54] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, Berlin, Heidelberg, 1999. Springer-Verlag.

[55] Dayeol Lee, David Kohlbrenner, Kevin Cheang, Cameron Rasmussen, Kevin Laeufer, Ian Fang, Akash Khosla an Chia-Che Tsai, Sanjit Seshia, Dawn Song, and Krste Asanovic. Keystone Enclave: An Open-Source Secure Enclave for RISC-V. https://keystone-enclave.org/files/keystone-risc-v-summit.pdf, 2018. (Accessed: February 12, 2020).

[56] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 19–33, Washington, DC, USA, 2014. IEEE Computer Society.

[57] Teng Li, Vikram K Narayana, Esam El-Araby, and Tarek El-Ghazawi. GPU resource sharing and virtualization on high performance computing systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 733–742. IEEE, 2011.

[58] Tyng-Yeu Liang and Yu-Wei Chang. GridCuda: A Grid-Enabled CUDA Programming Toolkit. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 141–146, March 2011.

[59] Arm Limited. Introducing Arm TrustZone. https://developer.arm.com/technologies/trustzone. (Accessed: February 12, 2020).

[60] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Dkaniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, January 2018.

[61] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN transformations. Cryptology ePrint Archive, Report 2017/452, 2017. http://eprint.iacr.org/2017/452.

[62] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal Covert Channels on Multi-core Platforms. In *USENIX Security Symposium*, 2015.

[63] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 301–316, New York, NY, USA, 2014. ACM.

[64] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks Are Practical. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[65] netem. https://wiki.linuxfoundation.org/networking/netem, 2019. (Accessed: February 12, 2020).

[66] NVIDIA. CUDA Zone. https://developer.nvidia.com/cuda-zone. (Accessed: February 12, 2020).

[67] NVIDIA. Driving Innovation: Building AI-Powered Self-Driving Cars. https://www.nvidia.com/en-us/self-driving-cars/. (Accessed: February 12, 2020).

[68] NVIDIA. RISC-V Story. https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf. (Accessed: February 12, 2020).

[69] NVIDIA. GPUs and DSLs for Life Insurance Modeling. https://devblogs.nvidia.com/gpus-dsls-life-insurance-modeling/, March 2016. (Accessed: February 12, 2020).

[70] NVIDIA. Microsoft Sets New Speech Recognition Record. https://news.developer.nvidia.com/microsoft-sets-new-speech-recognition-record/, August 2017. (Accessed: February 12, 2020).

[71] NVIDIA. NVIDIA CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-runtime-api/stream-sync-behavior.html, 2017. (Accessed: February 12, 2020).

[72] NVIDIA. CUDA Toolkit Documentation (Streams). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams, 2018. (Accessed: February 12, 2020).

[73] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Sebastian Nowozin Aastha Mehta, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium*, 2016.

[74] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border Control: Sandboxing Accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 470–481, New York, NY, USA, 2015. ACM.

[75] Sreepathi Pai and Keshav Pingali. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 1–19, New York, NY, USA, 2016. ACM.

[76] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'14, 2014.

[77] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. *ACM Trans. Embed. Comput. Syst.*, 15(1):15:1–15:25, January 2016.

[78] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys*, 51(6), 2019.

[79] Jonathan Power, Mark D Hill, and David A Wood. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *HPCA*, 2014.

[80] C. Reano, A. J. Pena, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Orti. CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution. *20th Annual International Conference on High Performance Computing*, 0:1–10, 2012.

[81] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[82] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Symposium on Operating Systems Principles*, SOSP'11, pages 233–248. ACM, 2011.

[83] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[84] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. *CoRR*, abs/1905.05726, 2019.

[85] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009.

[86] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 32, March 2013.

[87] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. *ACM Transactions on Computer Systems*, 32(1), 2014.

[88] Erik Smistad, Thomas L. Falch, Mohammadmehdi Bozorgi, Anne C. Elster, and Frank Lindseth. Medical image segmentation on GPUs A comprehensive review. *Medical Image Analysis*, 20(1):1–18, 2015.

[89] Matthew J.A. Smith, Mikayel Samvelyan, and Tabish Rashid. Using AI to Solve Collaborative Challenges by Playing StarCraft. https://news.developer.nvidia.com/using-ai-to-solve-collaborative-challenges-by-playing-starcraft/. (Accessed: February 12, 2020).

[90] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *USENIX ATC*, USENIX ATC'14, pages 109–120. USENIX Association, 2014.

[91] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[92] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *ISCA*, 2014.

[93] Kun Tian, Yaozu Dong, and David Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-Through. In *USENIX ATC*, pages 121–132, 2014.

[94] Florian Tramè and Dan Boneh. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *International Conference on Learning Representations*, ICLR '19, 2019.

[95] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 645–658. USENIX Association, 2017.

[96] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[97] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security*, August 2018.

[98] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. PixelVault: Using GPUs for Securing Cryptographic Operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1131–1142, New York, NY, USA, 2014. ACM.

[99] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *ISPASS*, 2016.

[100] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted Execution Environments on GPUs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[101] Lan Vu, Hari Sivaraman, and Rishi Bidarkar. GPU virtualization for high performance general purpose computing on the ESX hypervisor. In *Proceedings of the High Performance Computing Symposium*, page 2. Society for Computer Simulation International, 2014.

[102] Vultr.com. https://www.vultr.com/products/cloud-compute/. (Accessed: November 2019).

[103] Samuel Weiser and Mario Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 261–268, New York, NY, USA, 2017. ACM.

[104] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Transparent accelerator migration in a virtualized GPU environment. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid, pages 124–131, 2012.

[105] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[106] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog Computing: Platform and Applications. In *ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, HotWeb, 2015.

[107] Hangchen Yu, Arthur M. Peters, Amogh Akshintala, and Christopher J. Rossbach. AvA: Accelerated Virtualization of Accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[108] Yan Zhai, Lichao Yin, Jeffrey S Chase, Thomas Ristenpart, and Michael M Swift. CQSTR: Securing Cross-Tenant Applications with Cloud Containers. In *ACM Symposium on Cloud Computing*, 2016.

[109] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2017.

[110] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *2012 IEEE Symposium on Security and Privacy*, May 2012.

[111] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. Vulnerable GPU Memory Management: Towards Recovering Raw Data from GPU. *PoPETs*, 2017(2):57–73, 2017.

[112] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. Understanding The Security of Discrete GPUs. In *Proceedings of the General Purpose GPUs*, GPGPU-10, pages 1–11, New York, NY, USA, 2017. ACM.

TimeCrypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control

Lukas Burkhalter[1], Anwar Hithnawi[1,2], Alexander Viand[1], Hossein Shafagh[1], Sylvia Ratnasamy[2]

[1]*ETH Zürich* [2]*UC Berkeley*

Abstract

A growing number of devices and services collect detailed *time series* data that is stored in the cloud. Protecting the confidentiality of this vast and continuously generated data is an acute need for many applications in this space. At the same time, we must preserve the *utility* of this data by enabling authorized services to *securely* and *selectively* access and run analytics. This paper presents TimeCrypt, a system that provides *scalable* and *real-time* analytics over large volumes of encrypted time series data. TimeCrypt allows users to define expressive data access and privacy policies and enforces it cryptographically via encryption. In TimeCrypt, data is encrypted end-to-end, and authorized parties can only decrypt and verify *queries* within their authorized access scope. Our evaluation of TimeCrypt shows that its memory overhead and performance are competitive and close to operating on data in the clear.

1 Introduction

Recent years have seen explosive growth in systems and devices that collect time series data and relay it to cloud-based services for analysis. This growth is only expected to accelerate with the proliferation of IoT devices, telemetry services, and improvements in data analytics. However, with this growth has come mounting concerns over data protection and data privacy [66]. Today, the public concern over data privacy and confidentiality is reaching new heights in light of the growing scale and scope of data breaches [17, 29, 56]. To grasp the extent of this issue, one can look at the number of data breaches reported under the new GDPR *obligation to notify*, which has already exceeded 65,000 in the first year [73].

Over the last decade, encrypted databases [60, 61, 63, 75, 81] have emerged as a promising solution to tackle the problem of data breaches. The approach of keeping data encrypted while in-use allows users to query encrypted data while preserving both confidentiality and functionality. Research in this domain has led to various encrypted database designs, including designs for key-value stores [25], batch analytics [60], graph databases [53], and relational databases [63, 81]. This motivates the following natural question: *can we enable encrypted data processing for time series workloads?*

Time series workloads come with unique performance and security requirements that existing encrypted data processing systems fail to meet:

(i) Scalability and Interactivity. Query processing over time series data must simultaneously scale to large volumes of data, support low-latency interactive queries, *and* sustain high write throughput. To meet these challenges several dedicated databases have been designed for time series workloads [12, 33, 42, 45, 50, 62, 78]. A key aspect of these systems is their use of in-memory indices that store aggregate statistics, enabling faster query response times and data summarization. As we discuss in §6/§7, prior work on encrypted data processing does not easily lend itself to maintaining these in-memory indices. The overhead of the crypto primitives in encrypted data processing needs to be negligible to meet the scaling, latency, and performance requirements associated with time series workloads.

(ii) Secure Sharing. A key challenge in modern systems is that privacy must co-exist with the desire to extract value from the data, which typically implies *sharing* data to be analyzed by third-party services [54]. Hence, a truly comprehensive approach to data protection must also comprise mechanisms for secure sharing of encrypted data. Sharing should also be *fine-grained* since it is undesirable and often unnecessary to give parties unfettered access to the data. Instead, users may want to *(1)* share only aggregated statistics about the data (e.g., avg/min/max), *(2)* limit the resolution at which such statistics are reported (e.g., hourly vs. per-minute), *(3)* limit the time interval over which queries are issued (e.g., only June 2019), *(4)* or a combination of the above. Moreover, the desired granularity and scope of sharing can vary greatly across users and applications. Hence, support for encrypted query processing must go hand-in-hand with access control that limits the scope of data that users might query. The sharing paradigm in data-stream systems is distinctly different than in conventional databases. Data-stream settings feature a multitude of data sources continuously pushing data to the cloud, where various services that are often not known in advance can subscribe to consume and analyze data streams. Therefore, such systems require flexible access policies. Frequently, there is a need to fuse and analyze data from different sources collectively; this implies that we need to devise an end-to-end encryption scheme that is compatible with this sharing paradigm.

TimeCrypt. In this paper, we present TimeCrypt, a system that augments time series databases with efficient encrypted data processing. TimeCrypt provides cryptographic means to restrict the query scope based on data owners' defined policies. With TimeCrypt, data owners can cryptographically restrict user A to query encrypted data at a defined temporal range and granularity, while simultaneously allowing user B to execute queries on the same data at a different granularity without *(i)* introducing ciphertext expansion or data redundancy, *(ii)* introducing any noticeable delays, or *(iii)* requiring a trusted entity to facilitate this.

In this work, we introduce a partially homomorphic-encryption-based access control construction (HEAC) that supports both fine-grained *access control* and *computations* over encrypted data within a *unified scheme*. These two aspects have traditionally been addressed independently: the former through cryptographically enforced access control schemes [35, 39, 49, 59, 83] and the latter through encrypted data processing [60, 63, 75, 81]. HEAC simultaneously supports both while meeting the performance and access control requirements of time series workloads. A key insight behind the design of HEAC is based on the observation that time series data streams are continuous and time is the natural attribute for accessing and processing this data. Hence, we discretize data streams into fixed-length time segments, and encrypt each segment with a different key using symmetric-key homomorphic encryption. This allows us to express fine-grained access policies at the stream segment granularity. This, however, raises two challenges; we need to manage a large number of keys in an efficient and scalable manner and translate stream access policies to the corresponding keys succinctly. To overcome these challenges in HEAC, we associate keys with temporal segments. With this, we avoid the need to maintain a mapping between keys and ciphertexts. We derive these keys from a hierarchical key-derivation tree construction that allows us to express fine-grained access policies over stream data and share keys efficiently (i.e., with logarithmic complexity).

We provide an implementation and evaluation of a prototype of TimeCrypt on top of Cassandra. We evaluate TimeCrypt in a range of scenarios combining IoT devices, AWS (for data storage and processing), and time series traces from real-world applications. We show that TimeCrypt can support a wide range of applications by developing four applications which vary in complexity and scalability requirements. Finally, we show that TimeCrypt's performance is competitive with the baseline (plaintext) and it outperforms prior work by a factor of 2 to 52 (§6). Considering an ingest workload with 5.77 million data points per second *on a single machine*, TimeCrypt's throughput is reduced only by 2.9% for both data ingest and statistical queries over encrypted data.

Contributions. In summary, our contributions are:

- We introduce HEAC, an encryption-based access control construction for stream data that is additively homomorphic. HEAC additionally provides verifiable computations over ciphertexts to ensure the integrity of the outsourced encrypted computation.
- We design, implement, and evaluate TimeCrypt, the first scalable privacy-preserving time series database that meets the scalability and low-latency requirements associated with time series workloads. We introduce a design that protects the data confidentiality, yet maintains its utility by efficiently supporting a rich set of functionalities and analytics that are key to time series data. TimeCrypt supports data lifecycle operations such as ongoing data summarization and deletion that are common in time series databases. TimeCrypt supports expressive data access and privacy policies, enforceable by encryption.
- We make TimeCrypt's code publicly available[1], both as a standalone system and as a library to be integrated with other time series databases.

2 Overview

TimeCrypt achieves its competitive performance through a careful design of cryptographic primitives tailored for time series data workloads. To understand the rationale behind our techniques, we start this section by presenting relevant background on time series data, then we give an overview of TimeCrypt, and describe our security model.

2.1 Background on Time Series Data

Time series Applications. Time series data is increasingly prevalent across a wide range of systems (e.g., monitoring, telemetry, IoT) in diverse domains such as health, agriculture [82], transportation [69], operational insight [2], and smart cities. The growth of time series data is largely attributed to the rising demand for instrumentation. Individuals and organizations are continuously logging various metrics which report the state of systems or organisms for better diagnoses, forecasting, decision making, and resource allocation. The ability to capture and analyze this data in a timely manner is key for automation and is enabling a whole new spectrum of applications [2, 33, 40, 69, 82]. The proliferation of time series data has been coupled with increasing demand for high-performance analytics over large volumes of time series, and has led to numerous designs for databases that are optimized for time series workloads [12, 33, 42, 45, 50, 62, 78].

Time series Workloads. *(i) Write and Read:* Data is append-only and typically generated at an extremely high rate (high velocity) and is initially stored at a high resolution (large volume) [12, 62]. It is not unusual for applications in this space to report hundreds of millions of data points per day [7, 12].

[1] Available at: https://timecrypt.io/

Hence, sustaining high read and write throughputs and scalability are key requirements when storing and processing time series data. Time is the primary dimension for accessing and processing data. Queries primarily consider temporal ranges (e.g., values from the last 3h) rather than targeting individual points. *(ii) Analytics:* Queries are primarily of aggregate and statistical nature, and specialized indices for accelerating statistical queries are common in time series databases [12, 42, 65]. Additionally, analytics of diagnostic (e.g., anomaly or trend detection) and predictive nature (e.g., forecasting) are common in this space. *(iii) Data decay:* Time series data is often machine generated, continuous, and massive. Simultaneously, the value and relevance of data decays rapidly with time. Analytics largely favor recent data over older, and roll up aggregation is commonly applied to older data to reduce storage requirements. Hence, data retention and summarization [7, 62] are crucial for these systems.

The goal of our work is to retain the performance, functionality, and scalability of existing time series databases while augmenting them with strong security and privacy guarantees.

2.2 Architecture

TimeCrypt's architecture is analogous to that of conventional times-series databases [10, 11, 12, 42, 50], where a standard distributed key-value store is extended with additional logic for time series workloads. TimeCrypt includes a trusted *client library* to realize end-to-end encryption paired with access control and integrity verification. TimeCrypt consists of two components (i.e., the client and server libraries) and involves four parties (i.e., data owner, data producer, data consumer, and database server), as illustrated in Fig. 1. A *data producer* is an entity (e.g., IoT device) that generates and uploads time series data, and runs TimeCrypt's client library which handles stream preprocessing and encryption. The *data owner* can express access permissions to its generated data. Meanwhile, *data consumers* are entities (e.g., services) that are authorized to access a user's data to provide added value, such as visualizations, monitoring, and diagnoses. TimeCrypt's server executes statistical and analytical queries directly on encrypted data. TimeCrypt supports a rich set of foundational queries that are widely used in time series workloads (§4), i.e., statistical queries (e.g., min/max/mean), analytics (e.g., prediction, trend detection), and lifecycle operations (e.g., ongoing data summarization, deletion). The server builds *in-memory encrypted indices* to support fast queries and analytics (§4).

2.3 Goals for Stream Data Access Control

Encryption is an effective tool for protecting data from external threats, breaches, or malicious providers. However, a truly comprehensive approach to data protection must also include mechanisms for enforcing access control policies, to support the privacy and security principles of least privilege and data minimization, where data is protected by limiting unnecessary

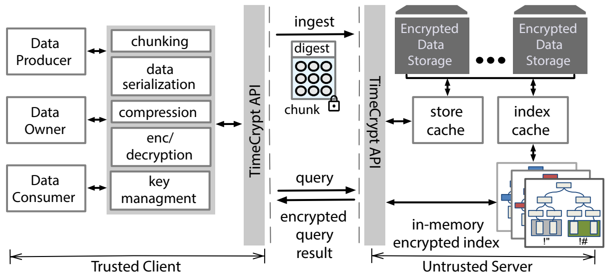


Figure 1: TimeCrypt's architecture.

exposure. State-of-the-art relational databases have security mechanisms designed for this purpose. The most adopted approach to support access control is based on views[2] and row-level access policies. However, specifying effective access control policies necessitates taking into consideration the semantics of data. Therefore, we investigated the major state-of-the-art time series databases [1, 4, 12, 42, 58, 65, 78] to understand the state of affairs in stream data access policies. We found that the only access policy restriction provided at the database interface, if any, is at the stream unit (i.e., grant or decline access to the entire stream). This binary protection level is however too coarse. This prompted the following question: What type of policies can offer the fine-grained protection that is required for selective and secure sharing of data streams? Stream data access control literature [24, 71] and time series applications designed for multi-user settings[5, 40] both recognize that policies which are expressed in time, resolution, and attributes are ideal for fine-grained access restrictions on streams. Examples of such policies can be a user choosing to simultaneously share hourly averages of their measured heart rate with their doctor and per-minute averages with their trainer but only for the duration of their workout session. Similarly, a datacenter operator might share resource utilization levels with a tenant but only for the duration of her job. Our goal is to translate these stream-specific sharing semantics into a cryptographically enforceable access control mechanism.

2.4 Threat Model

Our goal is to maintain the confidentiality and integrity of computations running on a cloud infrastructure that is potentially subject to an adversary that can read and tamper with data and manipulate query execution. In order to support sharing, we require a public-key infrastructure, such that entities can be identified and that a private/public key-pair can be associated with them. TimeCrypt provides the following guarantees in this setting:

Confidentiality. Data is encrypted using semantically secure encryption before it leaves the client device. Since decryption keys are never disclosed to the cloud provider, data confidentiality is guaranteed even in the case of a system compromise

[2] A view; also referred to as virtual table, is a dynamic window of a subset of the rows and columns in a database.

or malicious provider. Note that we do not employ property-revealing encryption, avoiding their inherent information leakage issues [55]. Our cryptographic access control mechanism ensures that data consumers can only query and access data according to the access policies defined by the data owner.

Integrity. TimeCrypt's integrity protection guarantees that, if a query completes, its output is equivalent to a correct execution on a trusted platform. Therefore, a malicious server cannot affect the computation, except by denying service. Note that even in case an integrity key for a stream is leaked, confidentiality remains intact.

Access Patterns. Similar to previous work [60, 63, 74, 81], TimeCrypt is non-oblivious, i.e., it does not protect against access pattern-based inferences in a trade-off for performance and scalability. Therefore, an adversary can learn which data the consumers are authorized to access by observing access patterns. TimeCrypt could be complemented with Oblivious RAM approaches [67] to hide these access patterns.

Access Control Collusion. Resolution based sharing in combination with interval sharing is not collusion resistant, even when considering a plaintext system. For example, any entity with access to aggregation over the intervals $[t_0, t_2)$ and $[t_1, t_2)$, can trivially derive the aggregation $[t_0, t_1)$ over the overlapped range by computing the difference. Hence, clients must be careful when sharing different resolutions over overlapping intervals. Furthermore, TimeCrypt comes with a trade-off between performance and collusion resistance when sharing non-continuous intervals. In the default mode, an adversary with access to two non-continuous intervals can compute the aggregation between the two intervals. For cases where this poses a privacy risk to applications, TimeCrypt provides a mechanism to prevent such collusion (§3.1) at the cost of increased decryption time.

A full security analysis with formal definitions and proofs can be found in the appendix of the extended version of the paper [23].

2.5 TimeCrypt Approach

TimeCrypt is a new encrypted time series database design that meets the scalability and low-latency requirements associated with time series workloads. We propose a new approach for data stream encryption that supports processing over encrypted data streams, computation integrity, and powerful access control within a unified scheme.

Data Abstraction. TimeCrypt stores data points in a stream as time-ordered chunks of predefined time intervals, i.e., $[t_i, t_{i+1})$ with a fixed interval size $\Delta = t_{i+1} - t_i$. Each data chunk also includes an encrypted digest that consists of statistical summaries about the underlying data. The digest enables TimeCrypt to compute statistical queries over time ranges efficiently, as we discuss next. At the client, the chunks are encrypted with standard symmetric encryption while the digests are encrypted with HEAC.

Aggregatable Digests. As HEAC is additively homomorphic[3], it supports secure aggregation of ciphertexts. However, to support queries beyond sum, we leverage *aggregatable encoding techniques* that exist in literature to support sophisticated statistical and analytical queries over encrypted data. At a high level, we introduce a per-chunk digest, which holds a vector of encoded values $\{x_0, ..., x_n\}$ that are encrypted with HEAC. To process queries, the server computes the aggregate function on the encrypted encodings across different digests. With this, we can support statistical queries that are inherently aggregation-based (e.g., sum, mean) or can be transformed to be aggregation-based (e.g., min/max, regression) (§4).

Encryption and Access Control. A key aspect of our scheme is tied to the observation that time series data streams are continuous. Consequently, to enable encrypted data processing that natively supports access control, we model data streams as a series of time segments, where each segment is encrypted with a different encryption key. We introduce a time-encoded keystream that maps keys to segments of the data stream, such that when a user restricts access to the data stream, only the corresponding range in the keystream is shared with the data consumer (§3.1). Based on the access policy, a data consumer is provided with the necessary decryption keys via an access tokens. Access tokens are encrypted with the data consumer's public key (hybrid encryption) and stored at the server. To enable sharing without enumerating all the keys and to support a *succinct key state*, we derive keys from a hierarchical tree key-derivation construction (§3.3). We also introduce a technique to support restricting access to a particular resolution level (§3.3.2), e.g., aggregated values at 10-minute resolution.

3 Encryption in TimeCrypt

In this section, we introduce the cryptographic components of TimeCrypt and present HEAC in more detail. HEAC, in essence is based on a symmetric homomorphic encryption [27]. However, we improve its performance by a factor of 2x for time series workloads by mapping keys to time and optimizing it for in-range ciphertext aggregations. Furthermore, we extend it to support fine-grained cryptographic access control capabilities tailored to time series data. Finally, we ensure computation integrity on encrypted data via Homomorphic Message Authentication Codes.

3.1 Symmetric Homomorphic Encryption.

We encrypt an integer m_i from the message space $[0, M-1]$ as $c_i = Enc_{k_i}(m_i) = m_i + k_i \ mod \ M$, with key $k_i \in [0, M-1]$. Given k_i, one can decrypt c_i as $Dec_{k_i}(c_i) = c_i - k_i \ mod \ M = m_i$. This scheme is semantically secure when the keys are pseudorandom and no key is reused [27].

[3] An additive homomorphic encryption scheme supports additions on ciphertexts, such that $decrypt(C_1 \oplus C_2) = decrypt(C_1) + decrypt(C_2)$.

Given the aggregated secret keys, one can decrypt the aggregated ciphertexts:

$$\sum_{i=0}^{n} m_i = Dec_{\sum_{i=0}^{n} k_i}(\sum_{i=0}^{n} c_i) = \sum_{i=0}^{n} c_i - \sum_{i=0}^{n} k_i \ mod\ M \quad (1)$$

We set M to 2^{64}, to support all integer sizes, without leaking any information about their original size.

Key Canceling. In the above scheme, the local computation to aggregate keys is linear in the number of aggregated ciphertexts, forcing the client to perform the same amount of computations as the server. We reduce this linear overhead to a constant, by leveraging the fact that time series data is generally aggregated in-range (i.e., over a contiguous range in time) as discussed in §2.1. We can therefore employ *key canceling* [6, 19, 31, 60]. This technique will also be relevant later, when we discuss integrity and access control (§3.2, §3.3). To enable this optimization, we choose the individual encryption keys such that the inner keys cancel each other out during aggregation. We do this by replacing the individual key k_i with a composite key that links subsequent messages:

$$Enc_{k'_i}(m_i) = m_i + k'_i \ mod\ M, \text{ with } k'_i = k_i - k_{i+1} \quad (2)$$

For decryption of an in-range aggregated ciphertext (Eq. 1), we now require only the two boundary keys:

$$\sum_{i=0}^{n} k'_i = (k_0 - \cancel{k_1}) + (\cancel{k_1} - \cancel{k_2}) \dots (\cancel{k_n} - k_{n+1}) \quad (3)$$

With key canceling, the decryption time in TimeCrypt is independent of the number of in-range aggregated ciphertexts. This scheme remains semantically secure [23, 27, 31]; an attacker without access to the keys cannot exploit the canceling property. However, when given access to keys for two non-continuous intervals, an adversary could learn aggregates about the skipped time between the two intervals. For example, when given access to $k_0, \dots, k_5$ and $k_{10}, \dots, k_{15}$, they could compute $\sum_{i=5}^{10} m_i \mod M$, given k_5 and k_{10}. The ramifications of this issue arise when users share adjacent intervals in the same stream with small gaps. TimeCrypt provides a hybrid key-canceling mechanism that limits this leakage in a trade-off for longer decryption times. We split the keys into epochs by replacing some k_i with non-canceling skip-keys k'_i, k''_i in $k_{i-1} - k_i$ and $k_i - k_{i+1}$, respectively. With this, we can share one interval per epoch without leakage. This increases the cost of aggregations over the epoch borders by two key derivations and one addition.

Time-Encoded Keystream. In TimeCrypt, access permissions are expressed with temporal ranges, e.g., Sep-14-15:00 till Sep-17-06:00 2019. Internally, TimeCrypt *chunks* data into fixed time segments of size Δ, which can be set per stream (e.g., 10 s intervals). In addition to the raw data points, each chunk is augmented with digests that are used for statistical query processing. Each chunk is encrypted with a fresh key from the keystream, indexed by the time window of the chunk. Assuming the data stream starts at timestamp t_0, the chunk digest m_i for the interval from t_i to t_{i+1} is encrypted as $c_i = Enc_{k_i - k_{i+1}}(m_i)$. By mapping keys to temporal ranges, a time range implicitly determines the position of the used key in the keystream. As a result, we sidestep the need to store identifiers of the keys along with the ciphertexts and avoid ciphertext expansion.

3.2 Integrity

Homomorphic encryption schemes are by design malleable, and therefore susceptible to ciphertext manipulation. In our setting, a dishonest server could try to drop, duplicate, or manipulate ciphertexts, resulting in incorrect query outputs. Incentives for deviations from the protocol could be as simple as trying to preserve resources by reducing the complexity of queries [72]. Beyond malicious behavior, integrity checks help to prevent faulty executions (e.g., data corruption, hardware faults, or misconfigurations). Ensuring computation integrity is essential, but is rarely considered in existing encrypted databases. Computation integrity can be achieved by requiring the server to provide a proof that the encrypted result was computed using the targeted data and function. Along this line, we introduce a verification protocol that allows the server to validate the output of in-range aggregations over ciphertexts with a succinct tag that can be verified in constant time at the client. To generate the proof, we use homomorphic Message Authentication Codes (HoMAC) [28]. While HoMACs have been introduced as cryptographic building blocks in the literature, existing solutions do not achieve integrity while maintaining scalability.

HoMAC. Conventional Message Authentication Codes (MACs) are small tags generated for each ciphertext which later ensure the authenticity and integrity of the ciphertext. *HoMACs* [9, 28] are conceptually similar to MACs, but additionally allow the server to perform computations like aggregations over the ciphertexts, and to produce new tags that authenticate the outputs of the computation. More precisely, the client generates a HoMAC tag σ for each ciphertext c and uploads (c, σ), where σ is defined as follows:

$$\sigma = HoMAC_s(c) = (s - c)/Z \mod p \quad (4)$$

where s is a per-ciphertext key, Z the HoMAC key, and p a prime number. The server computes aggregations on both the ciphertext and *HoMAC* tags $\sum_{i=0}^{n-1}(c_i, \sigma_i) = (c_{res}, \sigma_{res})$. The resulting tag σ_{res} authenticates and verifies that the output c_{res} corresponds to that specific aggregation. A client in possession of the *HoMAC* key material can verify the result by checking that the received σ_{res} tag matches the ciphertext c_{res}:

$$\sum_{i=0}^{n-1} s_i \overset{?}{=} c_{res} + \sigma_{res} Z \mod p \quad (5)$$

HoMACs are interesting for our use-case, since their symmetric nature makes them appealing to integrate with HEAC.

In contrast to authenticated data structures [46, 84], which can be used for outsourced computation verification, HoMAC tags do not need to be updated when new data is inserted. However, without further optimization, their verification overhead prevents their use in our setting.

Integrity Protocol. While HoMACs provide the desired integrity guarantees, they suffer from a verification overhead that is linear in the number of records in the aggregation query. Therefore, we apply a similar key canceling technique as already discussed above in the context of encryption: We define a *HoMAC* keystream $\{s_0, s_1, s_2, ...\}$ and, for each ciphertext c_i, the client computes the *HoMAC* tag σ_i as follows:

$$HoMAC_{s'_i}(c_i) = (s'_i - c_i)/Z = (s_i - s_{i+1} - c_i)/Z \bmod p \quad (6)$$

Setting $s'_i = s_i - s_{i+1}$ enables a constant time verification at the client side regardless of the input size, since only the two outer keys are required:

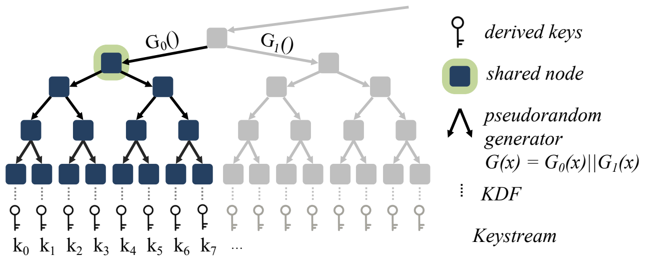
$$\sum_{i=0}^{n-1} s'_i = s_0 - s_n \overset{?}{=} c_{res} + \sigma_{res} Z \mod p \quad (7)$$

Using the key canceling concept in both encryption and integrity is a key enabler for our efficient cryptographic access control (§3.3). Since verification of aggregation results does not require access to the individual messages that were aggregated, our integrity protocol also integrates well with the resolution-based access control (§3.3.2).

HoMAC Security. For an attacker, it is computationally infeasible to generate a forged ciphertext and a tag which pass the verification. Note that we use different HoMAC key streams not just per-stream, but also per type of digest, i.e., target function. Therefore, the server cannot substitute a digest aggregation with another. In the case of key leakage, a party with access to the HoMAC key Z would be able to forge tags, but data confidentiality always remains intact. For a complete security treatment of HoMAC we refer to [28, 31] and the extended paper [23].

3.3 Cryptographic Access Control

The symmetric homomorphic encryption and HoMAC both require a pseudorandom keystream with one key for each message. The conventional approach to efficiently generating such keystreams would be to leverage a pseudorandom function with an initially exchanged secret key. This allows handling a large number of keys with one secret. However, with this approach, one could only share the entire data stream (i.e., all-or-none or in other words no fine-grained access control). Instead, we want to allow efficient sharing of arbitrary intervals, and want to allow users to restrict access to lower-resolution data, e.g., hourly or daily summaries. To realize this granular access control and to allow data owners to cryptographically enforce the scope of access to their data, we design a novel key derivation construction.



G0()
G1()
derived keys
shared node
pseudorandom generator G(x) = G0(x)||G1(x)
KDF
Keystream
k0 k1 k2 k3 k4 k5 k6 k7 ...

Figure 2: TimeCrypt's key derivation tree (leafs form a keystream).

3.3.1 Key Derivation Trees

Our key derivation is based on *key derivation trees*, i.e. balanced binary trees where each node contains a unique pseudorandom string. The leaf nodes represent the inputs to a key derivation function (KDF) to compute the keystream $\{k_0, k_1, k_2, ..., k_{2^h-1}\}$ as depicted in Fig. 2. The key derivation tree is built top-down from a secret random seed as the root. The child nodes are generated with a pseudorandom generator (PRG) that takes the parent string as the input. Our PRG consists of $G_0(x)$ for the left-hand child and $G_1(x)$ for the right-hand child, where x is the parent node. This procedure is applied recursively until the desired depth h in the tree is reached. We select a large h such that the keystream is virtually infinite, especially when considering that high-frequency streams will be chunked into e.g., one chunk per second. The pseudorandom generator can be realized from hash functions $G_0(x) = H(0||x), G_1(x) = H(1||x)$ with x as the key.

Access Token. The key derivation tree allows us to share segments of the keystream efficiently. Instead of sharing the segment key-by-key, the client shares a few inner tree nodes, combined into an *access token*. For instance in Fig. 2's toy example, a data owner grants access to the stream from t_0 to t_7, and the corresponding key segment $\{k_0, \dots, k_7\}$ is shared using a single node. In practice, a single node in the tree can be used to share thousands of keys. Note that given a node it is computationally not feasible (i.e., due to one-way property of PRGs) to compute the parent, sibling, or any of the ancestor nodes. Hence, a data consumer cannot compute any keys outside the segment they are granted access to.

Token Distribution. Once the data owner specifies an access policy for a data consumer, the TimeCrypt client generates an access token which encapsulates the inner nodes of the tree needed to derive the corresponding shared keystream segment specified in the access policy. We use the same key derivation tree for the encryption and HoMAC keystreams, but with a different KDF[4]. The token also contains encoded information about the subtree height and key identifier offset. TimeCrypt then encrypts the tokens with the data consumer's public key (i.e., hybrid encryption) and stores it at the server, such that the data consumer can fetch it to gain access to the keying

[4]Each leaf node of the primary key derivation tree is used to produce cryptographic keys needed for its corresponding chunk. Namely, keys for each element in the digest (i.e., query type), chunk, and HoMAC. Hence, we use different KDFs with the same node.

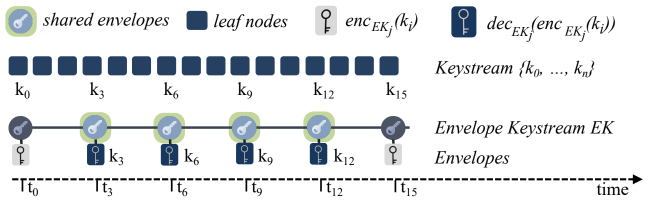


Figure 3: Envelope encryption for resolution-based access, showing envelopes required to share $[t_3, t_{12}]$ at a resolution of 3Δ.

material required to decrypt the data or query results. Note that TimeCrypt's key distribution is pluggable and we can employ alternative solutions. For instance, we can encrypt the token with attribute based encryption [83] to share tokens based on attributes (e.g., month as a key attribute).

3.3.2 Resolution-based Access Restriction

We now discuss how TimeCrypt provides crypto-enforced access control over the *resolution* at which data can be queried; i.e., the data owner not only restricts access to a time range per data consumer but also defines the temporal granularity (e.g., per minute) at which they can retrieve or query data.

Resolution Levels. In TimeCrypt, the highest resolution for queries and access control is defined by the chunk size Δ. Whenever we aggregate over an interval, we reduce the data resolution. For example, with one second chunks, an aggregation over 60 chunks results in a per-minute resolution. We can exploit the fact that keys cancel out during in-range aggregations, as described in §3.1, to cryptographically restrict access to lower resolution levels. In general, a ciphertext generated through an in-range aggregation over the time period $[t_i, t_j)$ has the form:

$$\sum_{x=i}^{j-1} c_x = \sum_{x=i}^{j-1} m_x + k_i - k_j \tag{8}$$

where the inner keys are canceled out. Hence, given access to just the boundary keys k_i and k_j, one can decrypt the aggregation, but none of the individual ciphertexts. Resolution levels must be multiples of the chunk size Δ and the segments at a given level must not overlap. Otherwise, data consumers could compute the difference of two aggregates overlapping by e.g., one chunk, allowing them to learn the data for that chunk which would violate the resolution-based access policy. For example, if the data owner wants to restrict access to a 3-fold resolution of the chunk size, the data owner would share only $\{k_0, k_3, k_6, \ldots\}$ with the data consumer. The data consumer can then decrypt the aggregated ciphertexts at the 3-fold (i.e., $3 \cdot \Delta$) or lower resolutions, but cannot access higher resolutions since the inner keys are missing.

Envelopes. While a data owner could share the boundary keys required for resolution-based access directly, this is not efficient since the number of keys necessary is linear in the length of the shared interval. Instead, the data producer stores the required boundary keys for a stream on the server, protected by another layer of encryption, the *envelope*. The keys used for the envelope encryption are derived from a new tree-based keystream $\{\bar{k}_0, \bar{k}_1, \bar{k}_2, \ldots\}$. For each resolution level, we use a different keystream for the envelope encryption. For example, if a data owner wants to make a per-minute resolution available for a stream with 20 s data chunks, the data owner encrypts the boundary keys $\{k_0, k_3, k_6, \ldots\}$ with the envelope keystream, and stores $\{enc_{\bar{k}_1}(k_0), enc_{\bar{k}_2}(k_3), enc_{\bar{k}_3}(k_9), \ldots\}$ on the server, as shown in Fig. 3.

Sharing a stream at a lower resolution is then again a matter of sharing a single access token, with the difference that the token now contains nodes of the key derivation tree for the envelope keystream, rather than for the original encryption keystream. A lower-resolution query returns, in addition to the encrypted result, two envelopes containing the two boundary keys required to decrypt the aggregated ciphertext. The overhead of resolution-based access control is similar to access control without resolution restrictions (§3.3), i.e., an access token consist of at most $O(log(n))$ nodes from the key derivation tree.

Dynamic Resolution Levels. In TimeCrypt, a user does not need to decide a priori on a fixed resolution for data consumers and can dynamically at any point in time define a new resolution. E.g., Alice can share her health data with a physician at minute-level (high-resolution) during physiotherapy from Jan-to-Feb, and from March reduce the resolution to hourly (low-resolution). The physician only sees high-resolution data for Jan-Feb and only hourly-data from March onwards.

3.4 Access Control Extensibility

Beyond temporal and resolution-based access policies, our construction also lends itself to enabling privacy policies on encrypted data, as combining ciphertexts from multiple users creates valid ciphertexts under a new *virtual aggregate key*. In the context of private operations, privacy policies permit a data consumer (e.g., analyst) to only run cross-stream aggregate queries, without having access to individual data streams. Similar to data access policies, privacy policies in our system are enforceable via encryption. As a concrete example, a user might want to allow a research lab to query her data but only if aggregated with a fixed set of n users, to preserve her individual privacy. Ensuring that a data consumer can only decrypt aggregates across a set of users can be realized by ensuring that she only has access to the *virtual aggregate key* (i.e., the data consumer never sees the keys for a particular user's stream in isolation). For instance, if a service is authorized to access an aggregate query over n encrypted messages from different users, then sharing only the *virtual aggregate key* $\sum_{i=1}^{i=n} k_i$ will ensure that the analyst can only decrypt the aggregated result. Therefore, we need a way to compute the *virtual aggregate key* without exposing the individual keys k_i of each user to any of the involved parties; the storage provider, authorized data consumer, or other users.

Function	Description
(1) `CreateStream(uuid, [config])`	Create a new stream, config defines parameters, e.g., chunk interval, operators.
(2) `DeleteStream(uuid)`	Delete specified stream with all associated data.
(3) `RollupStream(uuid, res,` [T_s, T_e]`)`	Rollup an existing stream or a segment of it to the specified resolution.
(4) `InsertRecord(uuid, [t, val])`	Serialize data points in a chunk and append to the end of the stream.
(5) `GetRange(uuid,` T_s, T_e`)`	Retrieve all data records within the specified time interval.
(6) `GetStatRange([uuid],` T_s, T_e`, resolution, [operators])`	Retrieve statistics for the given time interval and resolution, default [sum, count, mean, var, freq].
(7) `DeleteRange(uuid, start, end)`	Delete specified segment of the stream, while maintaining per-chunk digest.
(8) `GrantViewAccess(viewid, [princ-id])`	Grant access to an existing View.
(9) `CreateView(viewid, [policy])`	Create a View with the given policy in JSON format.
(10) `CheckView(viewid, princ-id)`	Retrieve a View token.

Table 1: TimeCrypt's basic API.

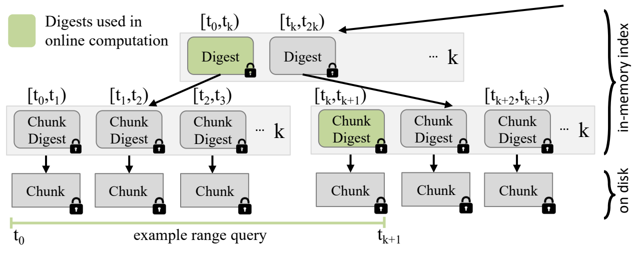


Figure 4: A statistical index for time series data with a *k*-ary time-partitioned aggregation tree. The pre-computed encrypted index allows for fast response times for statistical queries.

This can be accomplished by a secure aggregation protocol [6, 19] between the involved users and the analyst. The inputs to the protocol are the users' individual keys k_i and the output is the blinded contributions towards the *virtual aggregate key*. Queries across streams can be performed efficiently on the server, and the analyst can only decrypt the final result via a *virtual aggregate key*. In §6, we discuss a private crowdsourcing application atop of TimeCrypt that uses this technique.

4 Fast Analytics and Processing

To meet the requirements of time series databases, TimeCrypt must handle massive amounts of data, yet at the same time be able to serve queries with low latency. We address this challenge by introducing efficient client-side serialization/encryption and efficient encrypted indices on the server.

Client-side Data Serialization. The client serializes and encrypts data chunks containing the raw data, and digests. The content of a digest is set per stream based on the supported queries. The *default* query configuration of TimeCrypt supports *sum*, *count*, and *mean*. Other query types such as *variance*, *standard deviation*, *histogram*, bucket *min*/*max*, approximated *quantiles*, *trend detection*, and limited *filter* queries, can be enabled.

Server-side In-memory Encrypted Index. TimeCrypt's server maintains an in-memory encrypted index based on a time-partitioned aggregation tree over encrypted data. This is a key building-block that enables us to serve low-latency analytics on large encrypted data streams and enables efficient data retention. The index structure is a k-ary tree, where each internal node (digest) holds k statistical summaries of the subtree below it. The tree leaves store the chunk digests encrypted with HEAC at the client and represent the highest resolution data summaries (Fig. 4). On the arrival of a new chunk digest, the server inserts it as a leaf node, and updates statistical summaries of the parent nodes's by performing an encrypted aggregation. Any operation that can be expressed as an aggregation of the intermediate results from the child subtrees can be included in the summaries (see §4). Time series workloads are in-order and append-only, therefore updating the tree is straightforward. The encrypted index enables TimeCrypt to significantly decrease the response time for statistical queries, as the server avoids expensive serial scans. When executing a statistical range query over a time interval, the server traverses the tree and selects only the digests required to cover this interval, as illustrated in Fig. 4.

Statistical Queries. So far, we have developed the means to evaluate aggregates over ciphertexts, now we briefly[5] discuss how we combine aggregation with known encoding techniques [32, 47, 68] to allow TimeCrypt to compute more sophisticated statistics over ciphertexts. At a high level, each per-chunk digest holds a vector of encoded values that are encrypted with HEAC. For example, this vector might include the encrypted *sum* and *count* of the data points in the chunk. From this, we can then also calculate the *mean*. To compute quadratic functions, e.g., *var* and *stdev*, the vector includes the *sum of squares* of the points in the chunk. We can also include the *frequency count* of data points in the chunk, which yields valuable information to compute several statistical functions, such as *min, max, top N, bottom N, histograms, and quantiles.* For frequency counts, we use a vector $[c_{v_1},..,c_{v_n}]$, where each element in the vector c_{v_i} tracks the count of data points with value v_i. This works well for small n, which is often the case for (discrete) time series data. For larger ranges of values, we approximate the frequency count, i.e., each c_{v_i} tracks the count of a small range (*bin*) around v_i [32].

Advanced Analytics. In principle, any operations with aggregatable transformations can be supported in TimeCrypt, including a variety of sketch algorithms [52]. In addition, we can support many forms of machine learning, e.g., via aggregation-based encodings that allow private training of linear models [32, 47, 68]. These types of analytics are often employed in time series data to understand and detect runtime

[5]Due to space constraints, we keep the description here brief and refer to [32, 47, 68] for detailed description.

anomalies, trends, and patterns. We show how such analytics can be realized in TimeCrypt, using the example of *private trend detection*, i.e., identifying a general tendency over a defined time interval. It allows users to estimate the magnitude of a trend and is a highly related task to event detection (e.g., runtime anomalies). Linear regression using least-squares is a simple yet powerful method for trend detection [15]. To compute a linear regression model over a stream, the per chunk digest is defined as ($\sum_i x_i$, $\sum_i t_i x_i$) for $i \in [0, n)$. This way the expensive aggregations are done at the server. Such learning on summarized data also delivers privacy gains, as the raw data is not exposed in the training phase. In §6, we discuss the performance aspects of implementing such applications atop TimeCrypt.

Filter Queries. TimeCrypt supports filter queries with predefined predicates. One can define digest encodings that contain statistics over the values of the underlying chunk filtered with a predicate P (e.g., the sum of all values larger than 10). In the query phase, the filtered digests are used to compute statistics over the values matching the predicate P.

Time-Decayed Data Processing. As time series data ages, it is often aggregated into lower resolutions for long-term retention of historical data, while high-resolution data is aged-out. Typical strategies are based on compact summaries through aggregates [7, 43, 80]. TimeCrypt natively supports these approaches: as our index maintains aggregated summaries of the raw data, we can selectively delete aged-out raw data and prune lower nodes in the index. For example, we implement a retention policy based on the time-decayed merge algorithm [7] which keeps the data store compact (logarithmic in the input size) by dynamically re-compacting older data as new data arrives.

5 Prototype

API. TimeCrypt is realized as a service which exposes an interface similar to conventional time series stores [7, 12, 50]; applications can insert encrypted data, retrieve encrypted data by specifying an arbitrary time range, and process statistical queries over arbitrary ranges of encrypted data, as summarized in Table 1. In TimeCrypt, each stream is identified by a unique UUID and associated *stream metadata*, e.g., hostname, data type, sensor ID, location. Each stream has one writer (i.e., data producer) and one or multiple readers (i.e., consumers). A data owner can grant and specify access polices to consumers.

Granting Access to Stream Views. Data owners can manage access to their stream resources with the View API. *Views* define what a data consumer can access within the scope of the *View*. *Views* are set in JSON format, containing a unique identifier and a list of per stream access policies. In the current version, an owner can define the time range and the granularity that is accessible per stream, as for example: *"viewid": 2999, "streams": [{"uuid": [9,10], "from": "1546315200", "to": "1546315800", "granularity": "60s" }]*. This *View* defines an access scoped to stream 9 and 10 in the specified time window with a minute granularity. After the user defines a policy, the API assembles the access token with the necessary inner nodes of the key construction for the specified *View* (§3.3). The client library then derives a *View* key, encrypts the token along with the JSON description using AES-GCM and uploads it to the server. To give data consumers access to the *View*, the client invokes the *GrantViewAccess* command, which encrypts the *View* key with the respective consumer's public keys. The authorized consumers can download the tokens for the given *View* and can query the streams in the defined scope by the access policy. Though access policies are enforced by encryption, the intricacies of the key management are insulated from users in our design.

Reference Implementation. TimeCrypt's prototype is implemented in Java and consists of 6k SLOC with additional 4k SLOC for the applications and benchmark code. We used *Netty* [44] for network communication. TimeCrypt's server and client communicate over Google's *protobuffers* [37] protocol. The current prototype uses Cassandra [26] as the storage backend. The encrypted index is augmented with the in-memory cache *caffeine* [51] to speed up index node access. For the implementation of the cryptographic schemes, we used the Java security provider and a native C implementation of AES-NI. We compare the encrypted index performance with HEAC against alternative private aggregation schemes. We implemented three variants of the encrypted index based on Paillier [77] (Java *BigIntegers*), EC-ElGamal (OpenSSL [57]), and ASHE (we implemented it as described in [60]). Our code is available online.

6 Evaluation

In this section, we evaluate TimeCrypt's practicality. Our evaluation answers three core questions: *(1)* Can TimeCrypt meet the performance requirements of time series applications? *(2)* What are the performance gains of HEAC compared to alternatives? — HEAC supports access control and secure computation simultaneously; both aspects have traditionally been addressed with different schemes, consequently we examine alternatives independently in our evaluation. *(3)* Can TimeCrypt run compelling real-world applications?

Setup. Our experiments are conducted in Amazon AWS, on M5 instances equipped with a 2.5 GHz CPU running Ubuntu (16.04 LTS). TimeCrypt's server runs on an m5.2xlarge instance with 8 virtual processor cores (vcores) and 32 GB of RAM and a Cassandra node runs on an m5.xlarge instance with 4 vcores and 16 GB of RAM. The clients are simulated on several m5.xlarge instances. The client and server are located in the same data center network, with up to 10 Gbps bandwidth. In the microbenchmark, we quantify the overheads of encryption and decryption on end devices. We consider resource-constrained IoT devices; this class of devices is a major source of sensitive time series data. We use IoT OpenMotes (32-bit ARM M3 SoC 32 MHz) and a MacBook Pro 2.8 GHz Intel Core i7, with 16 GB RAM.

System	Micro	Index - Size	Average Ingest Time			Average Query Time (worst-case)		
	ADD	$1M$	$1k$	$1M$	$100M$	$1k$	$1M$	$100M$
TimeCrypt	1ns	8.1MB (1x)	10μs (1.7x)	16μs (1.3x)	22μs (1.3x)	21μs (1.6x)	46μs (1.3x)	50μs (1.1x)
TimeCrypt+	3ns	24.3MB (3x)	16μs (2.6x)	35μs (2.9x)	39μs (2.3x)	38μs (2.9x)	87μs (2.4x)	109μs (2.4x)
Plaintext	1ns	8.1MB (1x)	6μs (1x)	12μs (1x)	17μs (1x)	13μs (1x)	36μs (1x)	45μs (1x)

Table 2: Overview of evaluation results on the cloud, with 128-bit security, except for plaintext. The largest index size with 100M chunks, represents 50 billion data points in our health app.

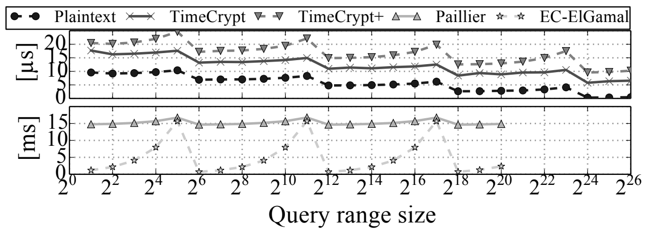


Figure 5: Aggregate queries over varying time ranges (i.e., query range size). Aggregating the entire index corresponds to retrieving the encrypted root.

We quantify the overhead of **TimeCrypt** (confidentiality) and **TimeCrypt+** (confidentiality plus query verification), and compare it to *(i)* operating on **plaintext** as the baseline, and *(ii)* **prior work** where we consider alternative encryption schemes for encrypting the digest, i.e., *Paillier* (used in CryptDB [63]), *EC-ElGamal* (used in Pilatus [74]) and *ASHE* (used in Seabed [60]). For access control, we compare to a strawman solution and a construction of KP-ABE (used in Sieve [83]), that we use to realize temporal access control similar to that supported by HEAC. Unless noted otherwise, we use 128-bit security [13], i.e., 3072-bit keys for Paillier and 256-bit elliptic curves for EC-ElGamal (i.e., prime256v1). For the microbenchmarks, we use synthetic large data that resembles the mhealth application (§6.4) dataset.

6.1 Encrypted Data Processing Performance

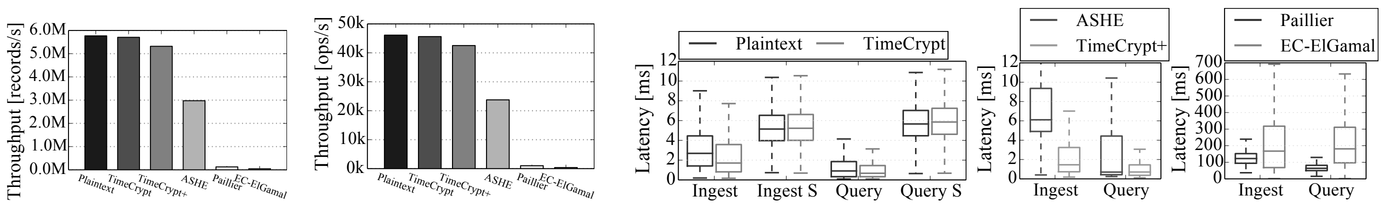
We now discuss the evaluation results of different aspects of the encrypted index, as summarized in Table 2. In the microbenchmark, the index supports one statistical operation (i.e., sum) for isolated overhead quantification, whereas in the E2E benchmark the index supports all our default queries. In all experiments, we instantiate 64-ary index trees and a keystream with one billion keys via the key derivation tree.

Index Size Expansion. To improve query efficiency, in-memory time series databases aggressively seek to reduce storage footprint, to support a model where almost all recent data can be stored in memory. When considering encryption for time series data, the degree of ciphertext expansion has a direct impact on the encrypted index storage footprint, hence impacting query efficiency. TimeCrypt has no ciphertext expansion for 64-bit values, TimeCrypt+ introduces a 128-bit expansion due to the HoMAC tag. The encryption schemes in prior work [63, 75, 81] exhibit large ciphertext expansion, e.g., for one million chunks we experience 96x index size expansion with Paillier. Hence, limiting the performance gains of in-memory processing and impacting query latency. ASHE [60] uses an encoding where the expansion depends on the order of aggregation. With in-range aggregation this amounts to 12.5% higher expansion compared to TimeCrypt.

Ingest Time. On each ingest, i.e., insertion of a leaf node, statistical aggregates of ancestor nodes are updated. In TimeCrypt, additions are as efficient as in *plaintext*. Hence, the average ingest time increases slightly due to the encryption cost; 1.3x for the large index. With verification the average ingest time increases by 3.2x due to the HoMAC overhead.

Query Performance. Fig. 5 shows the performance of the index for statistical range queries of different lengths, i.e., $[0, 2^x]$ with $x \in [0..26]$. As the length of queries increases fewer tree levels are traversed, which results in fewer cache fetches and lower computation time, e.g., the index depth of five is observable in Fig. 5. For plaintext and TimeCrypt the resulting pattern is similar due to the low cost of additions, while for TimeCrypt the decryption overhead is visible. Queries with non-power-of-k ranges require an index *drill down* on either end of the range. This increases the computation time logarithmic, $O(2(k\text{-}1)log_k(n))$ for a worst-case alignment, and not linear to the n stored chunks.

Comparison to Alternatives. In Fig. 6, we show HEAC's performance gains relative to the encryption schemes used in the other encrypted systems. For this experiment we launch an ingest/query workload, with one machine and 100 threads, where each thread constantly performs four statistical queries after each chunk ingest. The plaintext setting reaches a throughput of 5.77M records/s for ingest and 46.1k ops/s for statistical queries, as shown in Fig. 6a-b. TimeCrypt demonstrates an outstanding throughput for both ingest and statistical queries with only 2.9% slowdown compared to plaintext. With verification (TimeCrypt+), the slowdown increases to 7.8% due to the larger index size and HoMAC computations. TimeCrypt is by a factor of 2x, 20x, and 52x faster than ASHE, EC-ElGamal, and Paillier, respectively. Despite ASHE's lower encryption and decryption cost, the system throughput is by 2x lower due to the higher aggregation costs on the server. This is due to ASHE's key-encoding updates, which TimeCrypt eliminates with the time-to-key mapping. Fig. 6c-d shows the respective observed query latency. The impact of a small index cache (1 MB) is distinct, but similar for both plaintext and TimeCrypt, due to higher cache misses.



(a) Ingests (b) Statistical Queries (c) TimeCrypt vs. Plaintext (d) TimeCrypt+ & Alternatives

Figure 6: Latency and throughput for ingest and statistical queries for TimeCrypt with HEAC vs. EC-ElGamal, Paillier, & ASHE, and operating on plaintext indices. Heavy load experiment with a read-write ratio of 4 to 1, and also with extremely small (S) index cache (1 MB). The AWS load generator creates 1200 streams with 100 clients, corresponding to 48579 streams in our health app (Δ:10s, 50Hz data rate).

	HEAC		ASHE	Paillier
	[Enc/Dec]	[HoMAC]	[Enc/Dec]	[Enc/Dec]
IoT	1.08ms	20μs	0.3ms	1.59s / 1.62s
Laptop	5.1μs	0.2μs	1.5μs / 1.3μs	30ms / 15ms

Table 3: Performance of crypto operations with at least 80-bit security and 32-bit integers on IoT devices (OpenMote) vs. laptop (MacBook). TimeCrypt uses a key derivation tree with 2^{30} keys.

To compare how other encrypted systems perform while processing encrypted time series workloads, we run one aggregate query over one billion data records on CryptDB, Pilatus, Seabed, and TimeCrypt. Seabed requires seconds to process this query while CryptDB and Pilatus require minutes, whereas TimeCrypt can process such a query within a few milliseconds on a single machine.

6.2 Client Performance

Table 3 summarizes the enc/decryption and HoMAC costs of HEAC in TimeCrypt. TimeCrypt's cryptographic costs are dominated by the key derivation tree. Enc/decryption amount to 5.1 μs, which accounts for the time to compute the key. With HoMAC the clients incur 4% higher costs. To put this in prospective, this is three orders of magnitude faster than Paillier, EC-ElGamal, and ABE schemes with only few attributes. ASHE is faster in enc/decryption; the slight overhead in HEAC is due to the cost of deriving keys from our key derivation construction to support access control. Though overall, TimeCrypt is more performant in ingest and query performance due to its faster aggregations. The overhead of resolution-based access is defined by the access granularity. For instance, with 10 s chunk intervals and minute and hourly resolutions, the encryption cost increases by only 1% per day.

Low Power Devices. TimeCrypt is particularly compelling for battery-powered constrained devices used in the IoT and environmental sensing, where the power consumption of encryption is a serious challenge. Assuming one minute chunk intervals with TimeCrypt default queries, encryption consumes only 1.4% (400mJ) more battery per day on an OpenMote device compared to sending data in the clear.

6.3 Access Control

In the following, we look at the performance and scalability of our encryption-based access control mechanism. The overhead can be quantified as the cost of key distribution, deriving HoMAC and encryption keys, and computing the resolution envelopes. To characterize the overhead, we consider an example scenario where a data owner has 1000 streams and shares a subset of each stream with a data consumer.

Naïve Key Management. TimeCrypt realizes access control by encrypting units of stream data with unique keys. Consequently, efficient key distribution is important for the scalability of this approach. In a naïve approach, data owner can compile all the keys associated with the specified access policy and distribute the keys encrypted individually to each principle. However, this leads to access tokens of size $O(n)$ where n is the number of keys (i.e., units of stream data included in the access policy). With our key derivation construction, we have a logarithmic worst-case complexity in the number of shared stream units $O(log(n))$.

Communication. An access token in TimeCrypt contains in the worst-case $2(\log(n)-1)$ inner nodes of the tree key-derivation construction where n is the number of keys in the tree. This reduces the communication cost from a naïve approach from 50 GB to 1.28 MB, considering one year of data shared in our example scenario. With resolution-based access policies, the data consumer has to additionally download two envelopes per aggregation query (72 additional bytes).

Computation. Deriving the access token for all streams requires 145 ms. The decryption keys can be computed at a rate of 400k per second. With resolution-based access, the principal has to perform an additional decryption (for the envelope), which reduces the rate to 380k keys per second.

Storage. The storage cost can be broken down into two parts; key storage at the data consumer (1.28 MB), and resolution-related keying material at the server, which grows linearly with time (i.e., the envelopes). With a stream that consists of 10 s chunk intervals over one year with hour/day/month resolution support, the server stores 1.6 MB keying material (45.7k envelopes) per stream.

Comparison to an ABE-based Approach. Although, key-policy attribute-based encryption (KP-ABE) (used in Sieve [83]) is a powerful tool for access control, it comes

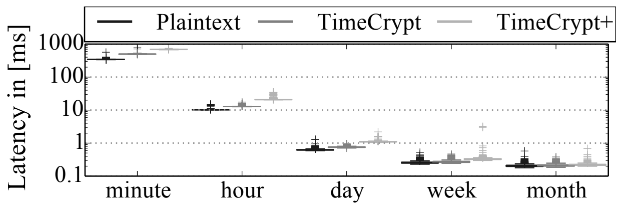
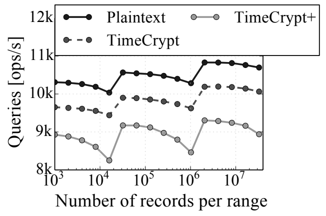
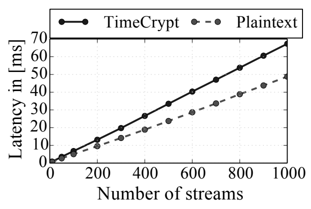


Figure 7: Latency in log-scale for statistical queries of one month data in our health app (121M records). The x-axis shows the granularity of the requested data from one minute to one month.



(a) DevOps Application



(b) SmartEnergy Application

Figure 8: Applications: (a) DevOps trend detection queries on CPU utilization over different number of records. (b) Energy consumption queries for a day over multiple streams in a smart meter application.

with a relatively high computational cost, especially for low-power devices and when used to enable fine-grained polices as needed in time series data. Compared to KP-ABE (implementation from [3]), HEAC is three orders of magnitude more efficient for encryption/decryption. For an IoT device encrypting one chunk per minute, an ABE-based solution drains one order of magnitude more battery life compared to HEAC. Additionally, ABE does not support computation on encrypted data.

Interrupt Key Canceling. TimeCrypt can add epoch borders to reduce the risk of leakage from aggregating the skipped interval between two shared non-continuous intervals (§3.1). Each additional epoch border within the query range incurs an additional computational cost to decryption (i.e., one key derivation and two additions). For example, considering a weekly epoch and a daily epoch in a data stream, the decryption cost for a monthly aggregate result increases by a factor of 2.5x and 14.5x, respectively. However, even for fine-grained epochs (e.g., over 300 per range), the decryption latency remains well below 1 ms and would not impact user perception.

6.4 Applications

In this section, we evaluate the end-to-end overhead of TimeCrypt and its effectiveness in running complex, real-world applications. We developed four apps atop of TimeCrypt that represent different challenging requirements and workloads.

mHealth Views - Interactivity. We implemented an mHealth dashboard for the Biovotion health tracker [18]. The dashboard shows summary plots of the underlying data (i.e, windowed AVG). The data consists of 12 different metrics at 50 Hz from the Biovotion sensor over two weeks, which we stretch to one year worth of data. Fig. 7 shows the response time for aggregation plots of last month's data (121M records). We also consider the extreme case of plotting one-month data at minute granularity (403 MB plot), which induces an overhead of 1.45x (2.0x for TimeCrypt+) in latency compared to plaintext. With lower granularity, the overhead sharply decreases and reaches 1.06x (1.29x for TimeCrypt+).

DevOps Trend Detection - Complex Analytics. We developed a trend detection app for CPU utilization. We use a CPU monitoring dataset generated by the time series benchmark suite [79] with 10 metrics, 10s data rate, and per minute chunk size Δ over one year. The results of a two-dimensional linear regression model on different ranges of an encrypted CPU monitoring stream are shown in Fig. 8a. TimeCrypt matches the plaintext performance (0.75% slowdown).

Smart Energy Service - Access Control Scalability. We extended a smart meter application, where a service computes the aggregated energy consumption per day over households. Each smart meter uploads a chunk every 5s, but the service can only compute per day aggregates for each stream. We use the ECO dataset [48], which contains smart meter data sampled at 1 HZ rate and collected over 8 months. Fig. 8b shows the query latency for the aggregated energy consumption over up to 1000 streams. TimeCrypt's overhead is attributed to multi-stream processing and resolution-based access. The overhead stems from the linearly increasing decryption costs in the number of streams that are aggregated.

Crowdsourced mHealth - Privacy Policy Transformation. We enhance the mHealth app with a crowdsourcing feature which enables users to opt-in their data to be part of crowdsourcing for a targeted research project, as described in §3.4. For *n* users, the secure aggregation protocol [19] adds a communication overhead of *n* Diffie-Hellman key exchanges per user to create the envelopes. The envelope enc/decryption increases linearly (e.g., below 1 ms for 100 users).

7 Related Work

There is a large body of research on privacy-preserving systems, encrypted search, and secure outsourced computation. For brevity, we focus our discussion here on works that are closest to TimeCrypt.

Encrypted Databases. Fuller et al. [34] provide a comprehensive overview of the encrypted database landscape. We now discuss several works in this space that are analogues to TimeCrypt. CryptDB [63] and Monomi [81] augment relational databases with encrypted data processing capabilities, however, encryption schemes used in these systems are not efficient enough to support interactive queries on large data. Seabed [60] focuses on Spark-like batch processing workloads and resorts to symmetric partial-homomorphic encryption to enable interactive queries on big data but without the tight latency requirements of time series data. CryptDB, Monomi, and Seabed do not support cryptographic access control or verifiable computation, as the case with TimeCrypt.

ENKI [41] and Pilatus [74] support sharing and encrypted computations but they scale poorly with the number of principals and the size of data. Also, they do not support fine-grained policies. Bolt [40] is an encrypted data storage system for time series data that supports retrieval of encrypted chunks but does not support server-side computation on encrypted data or fine-grained sharing. BlindSeer [61] enables private boolean search queries over an encrypted database by building an index with Yao's garbled circuits and primarily targets private search over large data with no support for statistical queries. It integrates access control for search queries but requires two non-colluding parties. Another line of research considers building data processing systems in trusted execution environments [14, 64, 72], which can provide confidentiality and integrity of queries. In TimeCrypt, we do not require dedicated hardware and rely on cryptographic primitives to ensure confidentiality and integrity of computation.

Cryptography-based Access. Cryptographically enforced access control is explored by crypto-systems [35] such as identity-based encryption, attribute-based encryption (ABE), predicate encryption, and functional encryption. They enable complex access control to encrypted data. ABE [8, 16, 38, 39, 59, 70] is the most expressive among them, though it comes with limitations with respect to fine-grained access and and dynamic updates [35]. Current ABE-based systems lack homomorphic capabilities (i.e., no computation on ciphertexts) and scalability required for time series data workloads. In general, adding homomorphic capabilities to ABE remains an open challenge [22]. Recently, important progress has been made on constructions of homomorphic attribute based encryption [20, 22, 30, 36]. However, they remain limited in functionality and are computationally expensive. A related line of work is searching over encrypted data with predicate evaluation [21, 76]. While predicate encryption schemes [21, 76] support range queries over encrypted data, they lack the required efficiency in our setting, as they require a linear scan through the database and also due to their underlying computationally expensive pairing-crypto.

8 Conclusion

In this paper, we presented TimeCrypt, a new scalable system that enables fast analytics over large encrypted data streams. TimeCrypt introduces HEAC, a novel encryption construction that enables execution of real-time analytics over encrypted stream data and empowers data owners to enforce access restrictions on encrypted data based on their privacy and access control preferences. Our evaluation on various large-scale workloads shows TimeCrypt's performance is close to operating on plaintext data, demonstrating the feasibility of providing high-performance and strong confidentiality guarantees when operating on large-scale sensitive time series data.

Acknowledgments

We thank our shepherd Ben Zhao, the anonymous reviewers, Amy Ousterhout, Scott Shenker, and Friedemann Mattern for their valuable feedback. This work was supported in part by the Swiss National Science Foundation Ambizione Grant, VMmware, Intel, and the National Science Foundation under Grant No.1553747.

References

[1] Amazon Kinesis Access Control. Online: https://docs.aws.amazon.com/streams/latest/dev/controlling-access.html.

[2] Datadog DevOps. Online: https://www.datadoghq.com/solutions/devops/.

[3] Fraunhofer-AISEC ABE Library. Online: https://github.com/Fraunhofer-AISEC/rabe.

[4] Graphite. Online: https://graphiteapp.org.

[5] Thingsboard. Online: https://thingsboard.io/.

[6] Gergely Ács and Claude Castelluccia. I Have a DREAM! (DiffeRentially privatE smArt Metering). In *International Workshop on Information Hiding*, pages 118–132. Springer, 2011.

[7] Nitin Agrawal and Ashish Vulimiri. Low-Latency Analytics on Colossal Data Streams with SummaryStore. In *ACM SOSP*, 2017.

[8] Shashank Agrawal and Melissa Chase. FAME: Fast Attribute-based Message Encryption. In *ACM CCS*, 2017.

[9] Shweta Agrawal and Dan Boneh. Homomorphic MACs: MAC-Based Integrity for Network Coding. In *ACNS*, pages 292–305. Springer, 2009.

[10] Amazon. Amazon Timestream. Online: https://aws.amazon.com/timestream/.

[11] Amazon. Time Series Processing. Online: https://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_timeseriesprocessing_16.pdf.

[12] Michael P. Andersen and David E. Culler. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *USENIX FAST*, 2016.

[13] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for Key Management Part 1: General (revision 3). *NIST special publication*, 2012.

[14] Andrew Baumann, Marcus Peinado, and Galen Hunt.

Shielding Applications from an Untrusted Cloud with Haven. *ACM TOCS*, 33(3):8, 2015.

[15] Julius S Bendat and Allan G Piersol. *Random Data: Analysis and Measurement Procedures*, volume 729. John Wiley & Sons, 2011.

[16] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-Policy Attribute-Based Encryption. In *IEEE Security and Privacy*, 2007.

[17] John Biggs. It's time to build our own Equifax with blackjack and crypto. Online. http://tcrn.ch/2wNCgXu, September 2017.

[18] Biovotion. Medical Grade Health Monitoring Wearable. Online: http://www.biovotion.com/.

[19] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *ACM CCS*, 2017.

[20] Dan Boneh, Craig Gentry, Sergey Gorbunov, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. Fully Key-Homomorphic Encryption, Arithmetic Circuit ABE, and Compact Garbled Circuits. Cryptology ePrint Archive, Report 2014/356, 2014.

[21] Dan Boneh and Brent Waters. Conjunctive, Subset, and Range Queries on Encrypted Data. In *TCC*, 2007.

[22] Zvika Brakerski, David Cash, Rotem Tsabary, and Hoeteck Wee. Targeted Homomorphic Attribute-Based Encryption. In *TCC*, 2016.

[23] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. [Extended Version] TimeCrypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control. *In ArXiv, arXiv:1811.03457v2 [cs.CR]*, 2020.

[24] Barbara Carminati, Elena Ferrari, and Kian Lee Tan. Specifying Access Control Policies on Data Streams. In *DASFAA*, pages 410–421. Springer, 2007.

[25] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: data structures and implementation. In *NDSS*, 2014.

[26] Cassandra database. Online: http://cassandra.apache.org/.

[27] Claude Castelluccia, Aldar CF Chan, Einar Mykletun, and Gene Tsudik. Efficient and Provably Secure Aggregation of Encrypted Data in Wireless Sensor Networks. *ACM TOSN*, 5(3):20, 2009.

[28] Dario Catalano and Dario Fiore. Practical Homomorphic MACs for Arithmetic Circuits. In *EUROCRYPT*, pages 336–352. Springer, 2013.

[29] Long Cheng, Fang Liu, and Danfeng Daphne Yao. Enterprise Data Breach: Causes, Challenges, Prevention, and future Directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(5), 2017.

[30] Michael Clear and Ciaran McGoldrick. Multi-identity and Multi-key Leveled FHE from Learning with Errors. In *CRYPTO*, 2015.

[31] Aloni Cohen, Shafi Goldwasser, and Vinod Vaikuntanathan. Aggregate Pseudorandom Functions and Connections to Learning. In *TCC*, 2015.

[32] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *USENIX NSDI*, 2017.

[33] Ketan Duvedi, Jinhua Li, Dhruv Garg, and Philip Ogden. Netflix: Scaling Time Series Data Storage. Medium, Online: https://medium.com/netflix-techblog/ec2b6d44ba39, January 2018.

[34] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. Sok: Cryptographically protected Database Search. In *IEEE Symposium on Security and Privacy*, 2017.

[35] W. C. Garrison, A. Shull, S. Myers, and A. J. Lee. On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud. In *IEEE Security and Privacy*, 2016.

[36] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *CRYPTO*, 2013.

[37] Google. Protocol Buffers. Online: https://developers.google.com/protocol-buffers/.

[38] Vipul Goyal, Abhishek Jain, Omkant Pandey, and Amit Sahai. Bounded Ciphertext Policy Attribute Based Encryption. In *ICALP*, 2008.

[39] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based Encryption for Fine-grained Access Control of Encrypted Data. In *ACM CCS*, 2006.

[40] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. Bolt: Data Management for Connected Homes. In *USENIX NSDI*, 2014.

[41] Isabelle Hang, Florian Kerschbaum, and Ernesto Damiani. ENKI: Access Control for Encrypted Query Processing. In *ACM SIGMOD*, 2015.

[42] InfluxDB. Online: https://www.influxdata.com/.

[43] InfluxDB Data Retention. Online: https://docs.influxdata.com/influxdb/v1.7/guides/downsampling_and_retention/.

[44] Java Netty Framework. Online: https://netty.io/.

[45] KairosDB. Online: https://kairosdb.github.io/.

[46] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *IEEE Security and Privacy*, 2016.

[47] Alan F Karr, Xiaodong Lin, Ashish P Sanil, and Jerome P Reiter. Secure Regression on Distributed Databases. *Journal of Computational and Graphical Statistics*, 2005.

[48] Wilhelm Kleiminger, Christian Beckel, and Silvia Santini. Household occupancy monitoring using electricity meters. In *ACM UbiComp 2015*, Osaka, Japan, 2015.

[49] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT. In *USENIX Security*, 2019.

[50] Florian Lautenschlager, Michael Philippsen, Andreas Kumlehn, and Josef Adersberger. Chronix: Long Term Storage and Retrieval Technology for Anomaly Detection in Operational Data. In *USENIX FAST*, 2017.

[51] Ben Manes. Coffein: Java Caching Library. Online: https://github.com/ben-manes/caffeine, 2018.

[52] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient Private Statistics with Succinct Sketches. In *NDSS*, 2016.

[53] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. Grecs: Graph encryption for approximate shortest distance queries. In *ACM CCS*, 2015.

[54] Richard Mortier, Jianxin Zhao, Jon Crowcroft, Liang Wang, Qi Li, Hamed Haddadi, Yousef Amar, Andy Crabtree, James Colley, Tom Lodge, and et al. Personal Data Management with the Databox: What's Inside the Box? In *ACM CAN*, 2016.

[55] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *ACM CCS*, 2015.

[56] Lily Hay Newman. The Alleged Capital One Hacker Didn't Cover Her Tracks. WIRED, Online: https://www.wired.com/story/capital-one-hack-credit-card-application-data/, July 2019.

[57] OpenSSL Software Foundation. Online: https://www.openssl.org/.

[58] OpenTSDB. Online: http://opentsdb.net/.

[59] Rafail Ostrovsky, Amit Sahai, and Brent Waters. Attribute-based Encryption with Non-monotonic Access Structures. In *ACM CCS*, 2007.

[60] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big Data Analytics over Encrypted Datasets with Seabed. In *USENIX OSDI*, 2016.

[61] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind Seer: A Scalable Private DBMSs. In *IEEE Security and Privacy*, 2014.

[62] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *VLDB*, 8(12):1816–1827, 2015.

[63] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *ACM SOSP*, 2011.

[64] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A Secure Database using SGX. In *IEEE Security and Privacy*, 2018.

[65] Prometheus. Online: https://prometheus.io/.

[66] Jingjing Ren, Daniel J. Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In *ACM IMC*, 2019.

[67] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security*, 2015.

[68] Alvin C. Rencher. *Linear Models in Statistics*. Wiley, 2007.

[69] Andrew Ross. The Connected Car Data Explosion: the challenges and opportunities. Information Age, Online: http://www.bbc.com/news/technology-42853072, January 2018.

[70] Amit Sahai and Brent Waters. Fuzzy Identity-Based Encryption. In *EUROCRYPT*, 2005.

[71] Sandeep Singh Sandha. StreetX: Spatio-Temporal Access Control Model for Data. *arXiv preprint arXiv:1711.03955*, 2017.

[72] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics

in the Cloud using SGX. In *IEEE Security and Privacy*, 2015.

[73] Mathew J. Schwartz. GDPR: Europe Counts 65,000 Data Breach Notifications So Far. Online: https://www.bankinfosecurity.com/gdpr-europe-counts-65000-data-breach-notifications-so-far-a-12489, May 2019.

[74] Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. Secure Sharing of Partially Homomorphic Encrypted IoT Data. In *ACM SenSys*, 2017.

[75] Hossein Shafagh, Anwar Hithnawi, Andreas Dröscher, Simon Duquennoy, and Wen Hu. Talos: Encrypted Query Processing for the Internet of Things. In *ACM SenSys*, 2015.

[76] Elaine Shi, John Bethencourt, T.-H.H. Chan, Dawn Song, and Adrian Perrig. Multi-Dimensional Range Query over Encrypted Data. In *IEEE Security and Privacy*, 2007.

[77] Brian Thorne. Java Paillier Library (javalier). Online: https://github.com/n1analytics/javallier, 2018.

[78] Timescale. Online: https://www.timescale.com/.

[79] Timescale. Time Series Benchmark Suite. Online: https://github.com/timescale/tsbs.

[80] Timescale Data Retention. Online: https://docs.timescale.com/v1.2/using-timescaledb/data-retention.

[81] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing Analytical Queries Over Encrypted Data. In *VLDB*, pages 289–300, 2013.

[82] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta N Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. FarmBeats: An IoT Platform for Data-Driven Agriculture. In *USENIX NSDI*, 2017.

[83] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds. In *USENIX NSDI*, 2016.

[84] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. IntegriDB: Verifiable SQL for Outsourced Databases. In *ACM CCS*, 2015.

Ghostor: Toward a Secure Data-Sharing System from Decentralized Trust

*Yuncong Hu, *Sam Kumar, and Raluca Ada Popa
University of California, Berkeley

Abstract

Data-sharing systems are often used to store sensitive data. Both academia and industry have proposed numerous solutions to protect the user privacy and data integrity from a compromised server. Practical state-of-the-art solutions, however, use weak threat models based on *centralized trust*—they assume that part of the server will remain uncompromised, or that the adversary will not perform active attacks. We propose Ghostor, a data-sharing system that, using only *decentralized trust*, (1) hides user identities from the server, and (2) allows users to detect server-side integrity violations. To achieve (1), Ghostor avoids keeping any per-user state at the server, requiring us to redesign the system to avoid common paradigms like per-user authentication and user-specific mailboxes. To achieve (2), Ghostor develops a technique called *verifiable anonymous history*. Ghostor leverages a blockchain *rarely*, publishing only a single hash to the blockchain *for the entire system* once every epoch. We measured that Ghostor incurs a 4–5x throughput overhead compared to an insecure baseline. Although significant, Ghostor's overhead may be worth it for security- and privacy-sensitive applications.

1 Introduction

Systems for remote data storage and sharing have seen widespread adoption over the past decade. Every major cloud provider offers it as a service (e.g., Amazon S3, Azure Blobs), and it is estimated that 39% of corporate data uploaded to the cloud is related to file sharing [51]. Given the relentless attacks on servers storing data [45], a long-standing problem in academia [14, 31, 35, 41, 49, 55, 60, 64, 75, 87] and industry [27, 46, 52, 77, 98] has been to provide useful security guarantees even when the storage server, and some users, are compromised by an adversary.

To address this, early systems [35, 48] have users encrypt and sign files. However, a sophisticated adversary can still:

- observe metadata about *users' identities* [24, 38, 47, 102]. Even if the files are encrypted, the adversary sees which users are sharing a file, which user is accessing a file at a given time, and the list of users in the system. Fig. 1 shows an example where the attacker can conclude that Alice has cancer from such metadata. Further, this allows the attacker to learn the graph of user social relations [81, 89].
- perform active attacks. Despite the signatures, an adversary can revert a file to an earlier state as in a *rollback attack*, or hide users' updates from each other as in a *fork attack*, without being detected. These are dangerous if, for example, the shared file is Alice's medical profile, and she does not learn that her doctor changed her treatment.

*Sam Kumar and Yuncong Hu contributed equally to this work. They are listed in alphabetical order by last name.

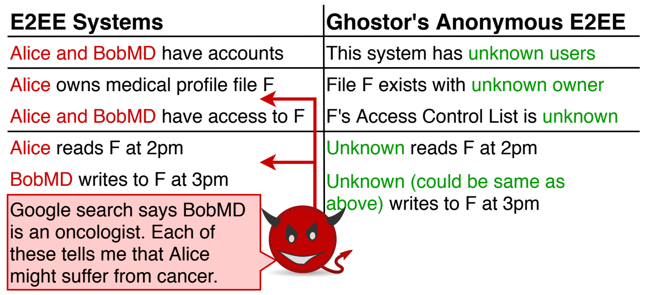

E2EE Systems
Ghostor's Anonymous E2EE
Alice and BobMD have accounts
This system has unknown users
Alice owns medical profile file F
File F exists with unknown owner
Alice and BobMD have access to F
F's Access Control List is unknown
Alice reads F at 2pm
Unknown reads F at 2pm
BobMD writes to F at 3pm
Unknown (could be same as above) writes to F at 3pm
Google search says BobMD is an oncologist. Each of these tells me that Alice might suffer from cancer.

Figure 1: An example of what a server attacker sees in a typical E2EE system versus Ghostor's Anonymous E2EE

Research over the past 15 years has striven to mitigate these attacks by providing *anonymity*—hiding users' identities from the storage server—or *verifiable consistency*—enabling users to detect rollback and fork attacks. In achieving these stronger security guarantees, however, state-of-the-art systems employ weaker threat models that rely on centralized trust: a trust assumption on a *few specific machines*. For example, they rely on a trusted party [66, 90], split the server into two components assuming one is honest [49, 54, 74], or assume the adversary is honest-but-curious (not malicious) [7, 16, 65, 104] meaning the attacker does not change the server's data or execution.

Attackers have notoriously performed highly targeted attacks, spreading malware with the ability to modify software, files, or source code [62, 106, 107]. In such attacks, a determined attacker can compromise any few *central servers*. Ideally, we would avoid *any trust* in the server or other clients, but unfortunately, that is impossible: Mazières and Shasha [69] proved that, if one cannot assume that clients are reliably online [55], clients cannot detect fork attacks without placing some trust in the server. Hence, this paper asks the question: Can we achieve strong privacy and integrity guarantees in a data-sharing system without relying on *centralized trust*?

To answer this question, we design and build Ghostor, an object store based on *decentralized trust* that achieves *anonymity* and *verifiable linearizability* (abbreviated VerLinear). At a high level, anonymity[1] means that the protocol does not reveal directly to the server any user identity with any request, as previously defined in the secure storage literature [54, 65, 74, 104]. As shown in Fig. 1, the server does not see which user owns which objects, which users have read or

[1]Outside of secure storage, *anonymity* is sometimes defined differently. In secure messaging, for example, an anonymous system is expected to hide the timing of accesses [97] and which files/mailboxes are accessed, but not necessarily the system's membership [26].

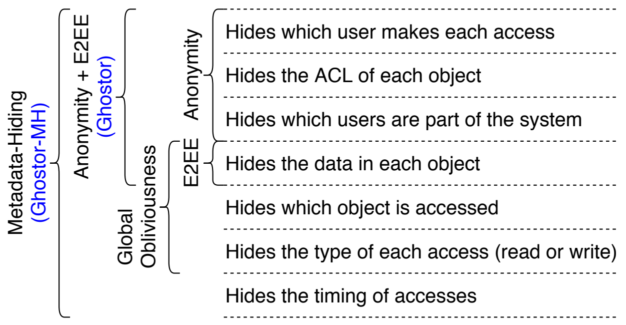

Metadata-Hiding
(Ghostor-MH)
Anonymity + E2EE
(Ghostor)
Anonymity
Global
Obliviousness
E2EE
Hides which user makes each access
Hides the ACL of each object
Hides which users are part of the system
Hides the data in each object
Hides which object is accessed
Hides the type of each access (read or write)
Hides the timing of accesses

Figure 2: Information leakage in a data-sharing system and associated privacy properties

write permissions to a given object, or even who are the users of the system. The server essentially sees **ghost**s accessing the **stor**age, hence the name "**Ghostor**." VerLinear means clients can verify that each write is reflected in later reads, except for benign reordering of concurrent operations as formalized by linearizability [42]. To achieve these properties, we build Ghostor's integrity on top of a consistent storage primitive based on decentralized trust, like a blockchain [17, 73, 105] or verifiable ledger [30, 44], while using it only *rarely*.

1.1 Hiding User Identities

Achieving anonymity in practical data-sharing systems like Ghostor is difficult because common system design paradigms, like user login, per-user mailboxes on the server, and client-side caching, let the server track users. We re-architect the system to avoid these paradigms (§4), using data-centric key distribution and encrypted key lists instead of server-side ACLs. Like prior systems [4, 33, 57], Ghostor uses cryptographic keys as capabilities, allowing the server and other users to verify each access is performed by an authorized user. Ghostor also leverages this technique to achieve anonymity by having all users authorized in a particular way *share* the same capability, and by distributing these capabilities to users without revealing ACLs to the server. We find this technique, *anonymously distributed shared capabilities*, interesting because anonymity is not typically a goal of public-key access control [4, 33] or capability-based systems [63, 72, 84].

An additional challenge is to guard against resource abuse while preserving anonymity. This is typically done by enforcing per-user resource quotas (e.g., Google Drive requires users to pay for additional space), but this is incompatible with Ghostor's anonymity. One solution is for users to pay for each operation via an anonymous cryptocurrency (e.g., Zcash [105]), but this puts an expensive blockchain operation in the critical path. To avoid this, Ghostor leverages blind signatures [18, 22, 23] to allow a user to pay the Ghostor server for service in bulk and in advance, while removing the linkage between payments and operations.

Relationship to obliviousness. Fig. 2 positions Ghostor's anonymity with respect to other privacy properties. Global obliviousness [7, 66], which hides which *object* is accessed across all uncompromised objects and users in the system, is orthogonal to Ghostor's anonymity, which hides which *user* performs each access. Obliviousness and anonymity are also complementary: (1) In some cases, without obliviousness, users may be identified based on access patterns. (2) Without anonymity, knowing which user issued a request may reveal information about what data that request may access. We view Ghostor's techniques for anonymity as a *transformation*:

- If applied to an E2EE system, we obtain **Ghostor, an anonymous E2EE system**.
- If applied to a globally oblivious scheme, we obtain **Ghostor-MH, a data-sharing scheme that hides all metadata** (except when initializing a group of objects or redeeming payments, as explained in Appendix D).

Hiding metadata from a malicious adversary, as in Ghostor-MH, is a very strong guarantee—existing globally oblivious schemes inherently reveal user identities [66] or assume the adversary is honest-but-curious [7, 65]. However, globally oblivious data-sharing schemes, like Ghostor-MH, are theoretical schemes that are far from practical. Thus, Ghostor-MH is only a proof of concept demonstrating the power of Ghostor's techniques to lift a globally oblivious scheme all the way to virtually zero leakage for a malicious adversary.

1.2 Verifiable Consistency

To provide VerLinear, prior work has clients sign hashes [55] so the clients can verify that they see the same hash, or store hashes on a separate hash server [49], trusted not to collude with the storage server. Neither technique can be used in Ghostor: client signatures are at odds with anonymity, and the hash server is a trusted party, which Ghostor aims to avoid.

One way to adapt the prior designs to Ghostor's decentralized trust is to store hashes on a blockchain, which can be accomplished by running the hash server in a smart contract. Unfortunately, this design is **too slow to be practical**. The client posts a hash on the blockchain for every object write, which is expensive: blockchains incur high latency per transaction, have low transaction throughput, and require cryptocurrency payment for each transaction [17, 73, 105].

To sidestep the limitations of a blockchain, we design Ghostor to only interact with the blockchain rarely and outside of the critical path. Ghostor divides time into intervals called *epochs*. At the end of each epoch, the Ghostor server publishes to the blockchain a small *checkpoint*, which summarizes the operations performed during that epoch for all objects and users in the system. Each user can then verify that the results of their accesses during the epoch are consistent with the checkpoint. The consistency properties of a blockchain ensure all clients see the same checkpoint, so the server is committed to a single history of operations and cannot perform a fork attack. Commit chains [53] and monitoring schemes [15, 93] are based on similar checkpoints, but Ghostor applies them to object storage while maintaining users' anonymity.

A significant obstacle is that a hash-chain-based history is not amenable to concurrent appends. Each entry in the history contains the hash of the previous entry, causing one

Goal	Technique
Anonymous user access control	Anonymously distributed shared capabilities (§4)
Anonymous server integrity verification	Verifiable anonymous history (§5)
Concurrent operations on a single object	Optimized `GET`s, two-phase protocol for `PUT`s (§5.4)
Anonymous resource abuse prevention	Blind signatures and proof of work (§6)
Hiding user IP addresses	Anon. network, e.g., Tor (§8)

Table 1: Our goals and how Ghostor achieves each one

operation to fail if a concurrent operation appends a new entry. Existing techniques for concurrent operations, such as SUNDR's VSLs [64], reveal *per-user* version numbers that would undermine Ghostor's anonymity. Our insight in Ghostor is to have the *server*, not the client, populate the hash of the previous entry when appending a new entry. To make this safe despite a malicious adversary, we carefully design a conflict resolution strategy, involving multiple *linked* entries in the history for each write, that prevents attackers from manipulating data via replay or time-stretch attacks.

We call the resulting design a *verifiable anonymous history*.

1.3 Summary of Contributions

Our goals and techniques are summarized in Table 1. Overall, this paper's contributions are:

- We design an object store providing anonymity and verifiable linearizability based only on *decentralized trust*.
- We develop techniques to (1) share capabilities for anonymity and distribute them anonymously, (2) create and checkpoint a verifiable anonymous history, and (3) support concurrent operations on a single object with a hash-chain-based history.
- We combine these with existing building blocks to instantiate Ghostor, an object store with anonymity and VerLinear.
- We also apply these to a globally oblivious scheme to instantiate Ghostor-MH, which hides nearly all metadata.

We also implemented Ghostor and evaluated it on Amazon EC2. Overall, Ghostor brings a 4-5x throughput overhead on top of a simplistic and completely insecure baseline. There are two types of latency overhead. Completing an individual operation takes several seconds. Afterward, it may take several minutes for a checkpoint to be incorporated into the blockchain, to confirm that no active attack has occurred for a batch of operations. We explain how these latencies play out in the context of a particular application, EHR Sharing (§7.1).

2 System Overview

Ghostor is an object store, which stores unstructured data items ("objects") and allows shared access to them by multiple users. We instantiate Ghostor as an object store (as in Amazon S3 or Azure Blobs) because it is a basic primitive on top of which more complex systems can be built. Fig. 3 illustrates Ghostor's architecture. Multiple users, with separate clients,

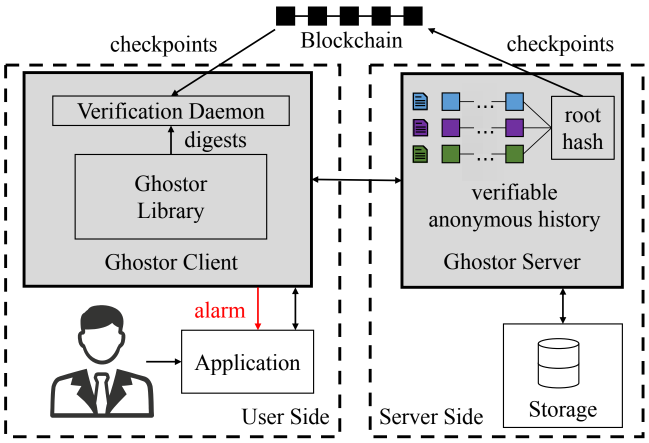


Figure 3: System overview of Ghostor. Shaded areas indicate components introduced by Ghostor.

have shared access to objects on the Ghostor server.

Server. The Ghostor storage server processes requests from clients. At the end of each epoch, the server generates a single small checkpoint and publishes it to the blockchain.

Client. The client software consists of a Ghostor library, linked into applications, and a verification daemon, which runs as a separate process. The Ghostor library receives requests from the application and interacts with the server to satisfy each request. Upon accessing an object, the library forwards a digest summarizing the operation to the verification daemon. At the end of each epoch, the daemon (1) fetches object histories from the server, (2) verifies that they are consistent with the server's checkpoint on the blockchain, and (3) checks that the digests collected during the epoch are consistent with the object histories, as explained in §5.

The daemon stores the user's keypair. If a user loses her secret key, she loses access to all objects that she created or was granted access to. Similarly, an attacker who steals a user's secret key can impersonate that user. To securely back up her key on multiple devices, a user can use standard techniques like secret sharing [82, 83, 99]. A user who accesses Ghostor from multiple devices uses the same key on all devices.

Application developers interact with Ghostor using the API below. Developers can work with usernames, ACLs, and object IDs, but Ghostor clients will not expose them to the Ghostor server. Below is a high-level description of each API call; a step-by-step technical description is in Appendix A.

◊ **create_user**(): Creates a Ghostor user by generating keys for a new user. This operation runs entirely in the Ghostor client—the server does not know this operation was invoked.

◊ user.**pay**(*sum*): Users pay the server through an anonymous cryptocurrency such as Zcash [105], and obtain *tokens* from the server proportional to the amount paid. These tokens can later be *anonymously redeemed* and used as proof of payment when invoking the below API functions.

◊ user.**create_object**(*id*): Creates an object with ID *id*, owned by user who invokes this. The client expends one token obtained from a previous call to **pay**. The *id* can be a meaningful name (e.g., a file path). It lives only within the client—the server receives some cryptographic identifier—so different clients can assign different *id*s to the same object.

◊ user.**set_acl**(*id*, *acl*): The user who invokes this must be the owner of the object with ID *id*. This function sets a new ACL for that object. For simplicity, only the owner of an object can set its ACL, but Ghostor can be extended to permit other users as well. The client encodes *acl* into an object header that hides user identities, as in §4. If new users are given access, they are notified via an out-of-band channel. Existing data-sharing systems also have this requirement; for example, Dropbox and Box send an email with an access URL to the user. In Ghostor, all keys are transferred in-band; the out-of-band channel is used only to *inform* the user that she has been given access. Ghostor does not require a specific out-of-band channel; for example, one could use Tor [29] or secure messaging [95,97].

◊ user.**get_object**(*id*), user.**put_object**(*id*, *content*): The user can GET or PUT an object if permitted by its ACL.

3 Threat Model and Security Guarantees

Against a malicious attacker who has compromised the server, Ghostor provides:

- verifiable linearizability, as described in §3.2, and
- a notion of user anonymity, described in §3.3: briefly, it does not reveal user identities, but reveals object access patterns. Ghostor-MH additionally hides access patterns.

Ghostor does not protect against attacks to availability. Nevertheless, its anonymity makes it more difficult for the server to selectively deny service to (or fork views of) certain users. Users, and the Ghostor client instances running on their behalf, can be malicious and can collude with the server.

Formal definitions and proofs for these properties require a large amount of space, so we relegate them to Appendix E and Appendix F. Below, we include only <u>*informal*</u> definitions.

3.1 Assumptions

Ghostor is designed to derive its security from decentralized trust. Thus, our threat model assumes an adversary who can compromise any few machines, as described below.

Blockchain. Ghostor makes the standard assumption that the blockchain is immutable and consistent (all users see the same transaction history). This is based on the assumption that, in order to attack a blockchain, the adversary cannot simply compromise a few machines, but rather a significant fraction of the world's computing power. Ghostor's design is not tied to a specific blockchain. Our implementation uses Zcash [105] because it supports both public and private transactions; we use Zcash's private transactions for Ghostor's anonymous payments. The privacy guarantees of Zcash can be implemented on top of other blockchains as well [11].

Network. We assume clients communicate with the server in a way that does not reveal their network information. This can be done using mixnets [21] or secure messaging [95,97] based on decentralized trust. Our implementation uses Tor [29].

3.2 Verifiable Linearizability

If an attack is immediately detectable to a user—for example, if the server fails to honor payment or provides a malformed response (e.g., bad signature)—we consider it an attack on *availability*, which Ghostor does not prevent.

Clients should be able to detect active attacks, including fork and rollback attacks. Some reordering of concurrent operations, however, is benign. We use *linearizability* [42] to define when reordering at the server is considered benign or malicious. *Informally*, linearizability requires that after a PUT completes, all later GETs return the value of either (1) that PUT, (2) a PUT that was concurrent with it, or (3) a PUT that comes after it. We provide a more formal definition in Appendix F. Ghostor provides *verifiable linearizability* (abbreviated *VerLinear*). This means that if the server deviates from linearizability, clients can detect it at the end of the epoch. We discuss how to choose the epoch length in §9. Ghostor does not provide consistency guarantees for malicious user, or for objects for which a malicious user has write access.

Guarantee 1 (Verifiable linearizability)**.** *For any object F and any list E of consecutive epochs, suppose that, for each epoch in E, the set of honest users who ran the verification procedure includes all writers of F in that epoch (or is nonempty if F was not written).* ***If*** *the server did not linearizably execute the operations that verifying clients performed in the epochs that they verified,* ***then*** *at least one of the verifying clients will encounter an error in the verification procedure and can generate a proof that the server misbehaved.*

3.3 Anonymity

As explained in §1.1, Ghostor's anonymity means that the server sees no user identities associated with any action. In particular, an adversary controlling the server cannot tell which user accesses each object, which users are authorized to access each object, or which users are part of the system.

Ghostor. We informally define Ghostor's privacy via a *leakage function*: what the server learns when a user makes each API call (§2). For **create_object** – **put_object**, the server learns the object identifier and the type of the operation. The server also sees the time of the operation, and the size of the encrypted ACL and encrypted object, which can be hidden via padding at an extra cost. **create_user** leaks no information to the server, and **pay** reveals only the sum paid and when. The server learns no user identities, no object contents, and no ACLs. If the attacker has compromised some users, he learns the contents of objects those users can access, including prior versions encrypted under the same key. Collectively, the verification daemons leak the number of clients performing verification for each object. If all clients in an object's ACL are honest and running, this equals the ACL size. If the ACL is padded to a maximum size, the owner should run verification more times to hide the ACL size. Ghostor does *not* hide access patterns or timing (Fig. 2). An adversary who uses

Keypair or Key	Description
(PVK, PSK)	Signing keypair used to set ACL
(RVK, RSK)	Signing keypair used to get object
(WVK, WSK)	Signing keypair used to put object
(OSK)	Symmetric key for object contents

Table 2: Per-object keys in Ghostor. The server uses the global signing keypair (SVK, SSK) to sign digests for objects.

this information cannot see the contents of files and ACLs because they are encrypted. But such an adversary could try to deduce correlations between which users issue different operations based on access patterns and timing, and in some cases, identify the user based on that information. This can be partially mitigated by carefully designing the application using Ghostor (§4.5). In contrast, Ghostor-MH does hide access patterns. In Appendix E, we formally define Ghostor's privacy guarantee in the simulation paradigm of Secure MPC.

Ghostor-MH. We informally define Ghostor-MH's privacy via a leakage function, as above. **create_object** reveals that a group of objects was created. **set_acl**, **get_object**, and **put_object** reveal nothing if the object's ACL contains only honest users; otherwise, they reveal which object was accessed. **create_user** and **pay** have the same leakage as described for Ghostor above. The leakage function also includes the total number of honest users in the system.

4 Hiding User Identities

System design paradigms used in typical data-sharing systems are incompatible with anonymity. We identify the incompatible system design patterns and show how Ghostor replaces them. Ultimately, we arrive at *anonymously distributed shared capabilities*, which allow Ghostor to enforce access control for anonymous users without server-visible ACLs.

4.1 No User Login or User-Specific Mailboxes

Data-sharing systems typically have some storage space on the server, called an *account file*, dedicated to a user's account. For example, Keybase [52] has a user account and Mylar [75] has a user mailbox where the user receives a key to a new file. Accesses to the account file, however, can be used to *link user operations*. As an example, suppose that when a user accesses an object, her client first retrieves the decryption key from a user-specific mailbox. This violates anonymity because the server can tell whether or not two accesses were made by the same user, based on whether the same mailbox was accessed first. Instead, Ghostor's anonymity requires that *any sequence of API calls (§2) with the same inputs, when performed by any honest user, results in the same server-side accesses*.

Ghostor does not have any user-specific storage as in existing systems. To allow in-band key exchange, Ghostor associates a *header* with each object. The object header functions like an object-specific mailbox, in that it is used to distribute the object's keys among users who have access to the object. Unlike a user-specific mailbox, it preserves anonymity because, for a given object, each user reads the *same* header

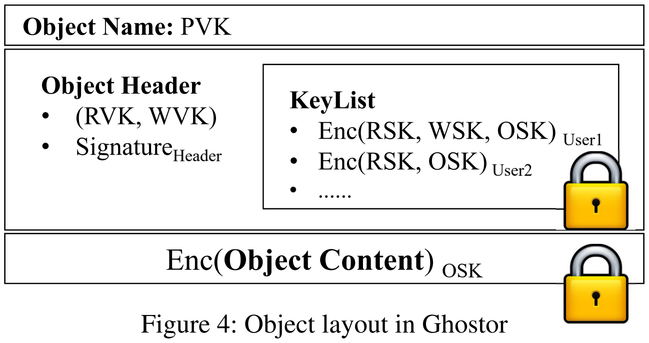


Figure 4: Object layout in Ghostor

before accessing it.

4.2 No Server-Visible ACLs

An honest server must be able to prevent unauthorized users from modifying objects, and users must be able to verify that objects returned by the server were produced by authorized writers. This is typically accomplished by having writers sign objects, and having the server check that the user who signed the object is on the object's ACL. However, this requires the ACL to be visible to the server, which violates anonymity.

We observe that by switching to a design based on *shared capabilities*, we can allow the server and other users to verify that writes are indeed made by authorized users, without requiring the server or other users to know the ACL of the object, or which users are authorized. Every Ghostor object has three associated signing keypairs (Table 2). All users of the object (and the server) know the verifying keys PVK, RVK, and WVK because PVK is the name of the object, and RVK and WVK are in the object header; the associated signing keys PSK, RSK, and WSK are *capabilities* that grant access to set the ACL, get the object, and put the object, respectively. To distribute these capabilities to users in the object's ACL, the owner places a *key list* in the object header. The key list contains, for each user, a list of capabilities encrypted under that user's public key. If a user has read/write access to an object, her entry in the key list contains WSK, RSK, and OSK; a user with only read access is given a dummy key instead of WSK. Crucially, different users with the same permission *share the same capability*, so the server cannot distinguish between users on the basis of which capability they use. When accessing an object, a user downloads the header and decrypts her entry in the key list to obtain OSK (used to decrypt the object contents) and her capabilities for the object.

Users sign updates to the object with WSK, allowing the server and other users to verify that each update is made by a user with write access. PSK is stored locally by the owner and is used to sign the header. The owner can set the object's ACL by (1) freshly sampling (RVK, RSK), (WVK, WSK), and OSK, (2) re-encrypting the object with OSK and signing it with WSK, (3) creating a new object header with an updated key list, (4) signing the new header with PSK, and (5) uploading it to the server. (RVK, RSK) will be relevant in §5.

Ghostor's object layout is summarized in Fig. 4.

4.3 No Server-Visible User Public Keys

Prior systems [64] reveal the user's public key to the server when the client interacts with it. For example, SUNDR requires users to provide a signature along with each operation. First, the signature itself could leak the user's public key. Second, to check the legitimacy of writes, the server needs to know the user's public key to verify the signature. The server can use the public key as a *pseudonym* to track users.

The key list in §4.2, however, potentially leaks users' public keys: each entry in the key list is a set of capabilities encrypted under a user's public key, but public-key encryption is only guaranteed to hide the message being encrypted, not the public key used to encrypt it. For example, an RSA ciphertext leaks which public key was used for encryption. Therefore, Ghostor uses *key-private* encryption [10], which is guaranteed to hide both the message and the public key.

In summary, Ghostor has users *share* capabilities for anonymity, and then distributes the capabilities anonymously, without revealing ACLs to the server. We call the resulting technique *anonymously distributed shared capabilities*.

4.4 No Client-Side Caching

Assuming that an object's ACL changes rarely, it may seem natural for clients to locally cache an object's keypairs $(\mathsf{RVK}, \mathsf{RSK})$ and $(\mathsf{WVK}, \mathsf{WSK})$, to avoid downloading the header on future accesses to that object. Unfortunately, the mere fact that a client did not download the header before performing an operation tells the server that the *same user* recently accessed that object. As a result, Ghostor's anonymity prohibits user-specific caching. That said, *server-side* caching of commonly accessed objects is allowed.

4.5 Careful Application Design

Ghostor does not hide access patterns or timing information from the server. A sophisticated adversary could, for example, deny or delay accesses to a particular object and see how access patterns shift, to try and deduce which user made which accesses. Therefore, one should carefully design the application using Ghostor to avoid leaking user identities in its access patterns. For example, just as Ghostor has no client-side caching or user-specific mailboxes, an application using Ghostor should avoid caching data locally to avoid requests to the server or using an object as a user-specific mailbox. Note that Ghostor-MH hides these access patterns.

5 Achieving Verifiable Consistency

Ghostor's *verifiable anonymous history* achieves the "verifiable equivalent" of a blockchain for critical-path operations, while using the underlying blockchain rarely. It consists of: (1) a hash chain of digests, (2) periodic checkpoints on a real blockchain, and (3) a verification procedure that does not require knowledge of user identities.

5.1 Hash Chain of Digests in Ghostor

We now achieve fork consistency for a single object in Ghostor using techniques inspired from SUNDR [64], but modified because SUNDR is not anonymous. Each access to an object, whether a `GET` or a `PUT`, is summarized by a *digest* shown in Table 3. The object's history is stored as a chain of digests.

Field	Description
Epoch	epoch when operation was committed
PVK, WVK, RVK	permission/writer/reader verifying key
$\mathsf{Hash}_{\mathsf{prev}}$	hash of previous digest in chain
$\mathsf{Hash}_{\mathsf{keylist}}$	hash of key list
$\mathsf{Hash}_{\mathsf{data}}$	hash of object contents
$\mathsf{Sig}_{\mathsf{client}}$	client signature with RSK, WSK, or PSK
$\mathsf{Sig}_{\mathsf{server}}$	server signature using SSK
nonce	random nonce chosen by client

Table 3: A digest for an operation in Ghostor

To access the object, a client first produces a digest summarizing that operation as in Table 3. This requires fetching the object header from the server, so that the client can obtain the secret key (RSK, WSK, or PSK) for the desired operation. Then the client fetches the latest digest for the object and computes $\mathsf{Hash}_{\mathsf{prev}}$ in the new digest. To `GET` the object, the client copies $\mathsf{Hash}_{\mathsf{data}}$ from the latest digest; to `PUT` it, the client hashes the new contents to obtain $\mathsf{Hash}_{\mathsf{data}}$. If the client is changing permissions, then $\mathsf{Hash}_{\mathsf{keylist}}$ is calculated from the new header; otherwise, it is copied from the latest digest.

Then the client signs the digest with the appropriate key and provides the signed digest to the server. The server signs the digest using SSK, appends it to a log, and returns the signed digest and the result of the operation. At the end of the epoch, the client downloads the digest chain for that object and epoch, and verifies that (1) it is a valid history for the object, and that (2) it contains the operations performed by that client. We specify protocol details in Appendix A.

Ghostor's digests differ from SUNDR in two main ways. First, for anonymity, a client does not sign digests using the user's secret key, but instead uses RSK, WSK, or PSK, which can be verified without knowing the user's public key. When inspecting the digest, the server no longer learns which user performed the operation, only that the user has the required permission. Second, each digest is signed by the server. Thus, if the server violates linearizability, the client can assemble the offending digests into a *proof of misbehavior*.

5.2 Checkpoint and Verification

The construction so far is susceptible to fork attacks [64], in which the server presents two users with different views over the same object. To detect fork attacks, Ghostor requires the server to produce a *checkpoint* at the end of each epoch, consisting of the hash of the object's latest digest and the epoch number, and publish the checkpoint to the blockchain. The *verification procedure* run by a client consists of fetching the checkpoint from the blockchain, checking it corresponds to the hash for the last digest in the list of digests obtained from the server, and running the verification in §5.1. The blockchain guarantees that all users see the same checkpoint. This prevents the server from forking two users' views, as the latest digests for two different views cannot both match the published checkpoint. In this way, we bootstrap the blockchain's consistency guarantees to achieve verifiable consistency over an entire epoch of operations.

5.3 Multiple Objects per Checkpoint

So far, the server puts one checkpoint in the blockchain *per object*, which is undesirable when there are many objects. We address this as follows. The server computes the hash of the final digest of each object, builds a Merkle tree over those hashes, and publishes the root hash in the blockchain as a single checkpoint for all objects. To verify integrity at the end of an epoch, a Ghostor client fetches the digest chain from the server for objects that are either (1) accessed by the client during the epoch or (2) owned by the client's user. It verifies that all operations that it performed on those objects are included in the objects' digest chains. Then, it requests Merkle proofs from the server to check that the hash of the latest digest is included in the Merkle tree at the correct position based on the object's PVK. Finally, it verifies that the Merkle root hash matches the published checkpoint.

Although we maintain a separate digest chain for each object, the collective history of operations, across all objects, is also linearizable. This follows from the classical result that linearizability is a local property [42]. Thus, Ghostor provides *verifiable linearizability across all objects, while supporting full concurrency for operations on different objects*.

5.4 Concurrent Operations on a Single Object

As explained in §5.1, the client must fetch the latest digest from the server to construct a digest for a new GET or PUT. If two clients attempt to GET or PUT an object concurrently, they may retrieve the same latest digest for that object, and therefore construct new digests that both have the same $\mathsf{Hash}_{\mathsf{prev}}$. An honest server can only accept one of them; the other operation must be aborted. A naïve fix is for clients to acquire locks (or leases) on objects during network round trips, but this limits single-object throughput according to client round-trip times. How can we allow concurrent operations on a single object without holding server-side locks during round trips? We explain our techniques at a high level below; Appendix A contains a full description of our protocol.

GETs. We optimize GETs so that clients need not fetch the latest digest, obviating the need to lock for a round trip. When a client submits a GET request to the server, the client need not include $\mathsf{Hash}_{\mathsf{prev}}$, $\mathsf{Hash}_{\mathsf{data}}$, or $\mathsf{Hash}_{\mathsf{keylist}}$ in the digest presented to the server. The client includes the remaining fields and a signature over only those fields. Then, the server chooses the hashes for the client and returns the resulting digest, signed by the server. Although the server can replay operations, this is harmless because GETs do not affect data. When the verification daemon verifies a GET, it checks the client signature without including $\mathsf{Hash}_{\mathsf{prev}}$, $\mathsf{Hash}_{\mathsf{data}}$, or $\mathsf{Hash}_{\mathsf{keylist}}$.

PUTs. The above technique does not apply to PUTs, because the server can roll back objects by replaying PUTs. Simply using a client-provided nonce to detect replayed PUTs is not sufficient, because the server can delay incorporating a PUT (which we call a *time-stretch* attack) to manipulate the final object contents. For PUTs, Ghostor uses a two-phase protocol. In the PREPARE phase, the client operates in the same way as GET, but signs the digest with WSK; the server fills in the hashes, signs the resulting digest, appends it to the object's digest chain, and returns it to the client. In the COMMIT phase, the client creates the final digest for the operation—omitting $\mathsf{Hash}_{\mathsf{prev}}$ and appending an additional field $\mathsf{Hash}_{\mathsf{prep}}$, which is the hash of the server-signed digest obtained in the PREPARE phase—and uploads it to the server with the new object contents. The server fills in $\mathsf{Hash}_{\mathsf{prev}}$ based on the object's digest chain (which could have changed since the PREPARE phase), signs the resulting digest, appends it to the object's digest chain, and returns it to the client. The server can replay PREPARE requests, but it does not affect object contents. The server cannot generate a COMMIT digest for a replayed PREPARE request, because the client signed the COMMIT digest including the hash of the server-signed PREPARE digest, which includes $\mathsf{Hash}_{\mathsf{prev}}$. The server can replay a COMMIT request for a particular PREPARE request, but this is harmless because of our conflict resolution strategy described below.

Resolving Conflicts. If two accesses are concurrent (i.e., neither commits before the other prepares), then linearizability does not require any particular ordering of those operations, only that all clients perceive the same ordering. If a GET is concurrent with a PUT (GET digest between the PREPARE and COMMIT digests for a PUT), Ghostor linearizes the GET as happening before the PUT. This allows the result of the GET to be served immediately, without waiting for the PUT to finish. For concurrent PUTs, it is unsafe for the linearization order to depend on the COMMIT digest, because the server could perform a time-stretch or replay attack on a COMMIT digest, to manipulate which PUT wins. Therefore, Ghostor chooses as the winning PUT the one whose PREPARE digest is latest. The server can still delay PREPARE digests, but the client can choose not to COMMIT if the delay is unacceptably large. To simplify the implementation of this conflict resolution procedure, we require that the PREPARE and COMMIT phases happen over the same session with the client, during which the server can keep in-memory state for the relevant object. This allows the server to match PREPARE and COMMIT digests without additional accesses to secondary storage.

Verification Complexity. To verify PUTs, the verification daemon must check that $\mathsf{Hash}_{\mathsf{data}}$ only changes on COMMIT digests for winning writes. Thus, it must keep track of all PREPARE digests since the latest PREPARE digest whose corresponding COMMIT has been seen. We can bound this state by requiring that PUT requests do not cross an epoch boundary.

ACL Updates. We envision that updates to the ACL will be rare, so our implementation does not allow **set_acl** operations to proceed concurrently with GETs or PUTs. It may be possible to apply a two-phase technique, similar to our concurrent PUT protocol, to allow **set_acl** operations to proceed concurrently with other operations. We leave exploring this to future work.

6 Mitigating Resource Abuse

To prevent resource abuse, commercial data-sharing systems, like Google Drive and Dropbox, enforce per-user resource quotas. Ghostor cannot do this, because Ghostor's anonymity prevents it from tracking users. Instead, Ghostor uses two techniques to prevent resource abuse without tracking users: anonymous payments and proof of work.

6.1 Anonymous Payments

A strawman approach is for users to use an anonymous cryptocurrency (e.g., Zcash [105]) to pay for each expensive operation (e.g., operations that consume storage). Unfortunately, this requires a separate blockchain transaction for each operation, limiting the system's overall throughput.

Instead, Ghostor lets users pay for expensive operations *in bulk* via the **pay** API call (§2). The server responds with a set of *tokens* proportional to the amount paid via Zcash, which can later be redeemed *without using the blockchain* to perform operations. Done naïvely, this violates Ghostor's anonymity; the server can track users by their tokens (tokens issued for a single **pay** call belong to the same user).

To circumvent this issue, Ghostor uses *blind signatures* [18, 22, 23]. A Ghostor client generates a random token and *blinds* it. After verifying that the client has made a cryptocurrency payment, the server signs the blinded token. The blind signature protocol allows the client to *unblind* it while preserving the signature. To redeem the token, the client gives the unblinded signed token to the server, who can verify the server's signature to be sure it is valid. The server cannot link tokens at the time of use to tokens at the time of issue because the tokens were blinded when the server originally signed them.

6.2 Proof of Work (PoW)

Another way to mitigate resource abuse is **proof of work (PoW)** [6]. Before each request from the client, the server sends a random $\mathsf{challenge}$ to the client, and the client must find a proof such that $\mathsf{Hash}(\mathsf{challenge}, \mathsf{proof}, \mathsf{request}) < \mathsf{diff}$. diff controls the difficulty, which is chosen to offset the amplification factor in the server's work. Because of the guarantees of the hash function, the client must iterate through different proofs until it finds one that works. In contrast, the server efficiently checks the proof by computing one hash.

6.3 Anonymous Payments & PoW in Ghostor

Ghostor uses anonymous payments and PoW together to mitigate resource abuse. Our implementation requires anonymous payment only for **create_object**, which requires the server to commit additional storage space for the new object. This is analogous to systems like Google Drive or Dropbox, which require payment to increase a user's storage limit but do not charge based on the count or frequency of object accesses. Implicit in this model are hard limits on object size and per-object access frequency, which Ghostor can enforce. Although our implementation requires payment only for **create_object**, an alternate implementation may choose to require payment for every operation except **pay**. Ghostor requires PoW for all API calls. This includes **pay** and **create_object**, to offset the cost of Zcash payments and verifying blind signatures.

7 Applying Ghostor to Applications

In this section, we discuss two applications of Ghostor that we implemented: EHR Sharing and Ghostor-MH.

7.1 Case Study: EHR Sharing

Our goal in this section is to show how a real application may interface with Ghostor's semantics (e.g., ownership, key management, error handling) and how Ghostor's security guarantees might benefit a real application. To make the discussion concrete, we explore a particular use case: multi-institutional sharing of electronic health records (EHRs). It has been of increasing interest to put patients in control of their data as they move between different healthcare providers [37, 43, 85]. As it is paramount to protect medical data in the face of attackers [28], various proposals for multi-institutional EHR sharing use a blockchain for access control and integrity [5, 70]. Below, we explore how to design such a system using Ghostor to store EHRs in a central object store, using only decentralized trust. We also implemented the system for Open mHealth [3].

Each patient owns one or more objects in the central Ghostor system representing their EHRs. Each patient's Ghostor client (on her laptop or phone) is reponsible for storing the PSKs for these objects. The PSKs could be stored in a wristband, as in [70], in case of emergency situations for at-risk patients. When the patient seeks treatment from a healthcare provider, she can grant the healthcare provider access to the objects containing the relevant information in Ghostor. Each healthcare provider's Ghostor client maintains a local *metadata database*, mapping patient identities (object IDs, §2) to PVKs. This mapping could be created when a patient checks in to the office for the first time (e.g., by sharing a QR code).

Benefits. Existing proposals leverage a blockchain to achieve integrity guarantees [5, 70] but use the blockchain more heavily than Ghostor: for example, they require a blockchain transaction to grant access to a healthcare provider, which results in poor performance and scalability. Additionally, Ghostor provides anonymity for sharing records.

Epoch Time. An important aspect of Ghostor's semantics is that one has to wait until the next epoch before one can verify that no fork has occurred. It is reasonable to fetch a patient's record at the time that they check in to a healthcare facility, but before they are called in for treatment. This allows the time to wait until the end of an epoch to overlap with the patient's waiting time. In the case of scheduled appointments, the record can be fetched in advance so that integrity can be verified by the time of the appointment. An epoch time of 15–30 minutes would probably be sufficient.

Error Handling. If a healthcare provider detects a fork when verifying an epoch, it informs other healthcare providers of the integrity violation out-of-band of the Ghostor system. Ghostor does not constrain what happens next. One approach, used in Certificate Transparency (CT), is to abandon the Ghostor server for which the integrity violation was detected. We envision that there would be a few Ghostor servers in the system, similar to logs in CT, so this would require affected users to migrate their data to a new server. Another approach is to handle the error in the same way that blockchain-based systems [5, 70] handle cases where the hash on the blockchain does not match the hash of the data—treat it as an availability error. While neither solution is ideal, it is better than the status quo, in which a malicious adversary is free to perform fork or rollback attacks undetected, causing patients to receive incorrect treatments based on old or incorrect data, potentially resulting in serious physical injury.

7.2 A Metadata-Hiding Data-Sharing Scheme

Ghostor's anonymity techniques can be combined with a globally oblivious scheme, AnonRAM [7], to obtain a *metadata-hiding* object-sharing scheme, *Ghostor-MH*. Ghostor-MH is *not* a practical system, but only a theoretical scheme; our goal is to show that Ghostor's techniques are complementary to and compatible with those in globally oblivious schemes. Below we summarize how we apply Ghostor's techniques in Ghostor-MH; we discuss Ghostor-MH in more detail in Appendix D. First, we apply Ghostor's principle of switching from a user-centric to a data-centric design. Whereas each ORAM instance in AnonRAM corresponds to a user, each ORAM instance in Ghostor-MH corresponds to an *object group*, a fixed-sized set of objects with a shared ACL. Second, we apply the design of Ghostor's object header in Ghostor-MH. This is accomplished by storing the ORAM secret state, encrypted, on the server. Finally, we use similar techniques to mitigate resource abuse in Ghostor-MH as we do in Ghostor.

8 Implementation

We implemented a prototype of Ghostor in Go. It consists of three parts, as in Fig. 3, server (≈ 2100 LOC), client library (≈ 1000 LOC), and verification daemon (≈ 1000 LOC), which all depend on a set of core Ghostor libraries (≈ 1400 LOC).

Our implementation uses Ceph RADOS [101] for consistent, distributed object storage. We use SHA-256 for the cryptographic hash and the NaCl secretbox library (which uses XSalsa20 and Poly1305) for authenticated symmetric-key encryption. For *key-private* asymmetric encryption (to encrypt signing keys in the object header), we implemented the El Gamal cryptosystem, which is *key-private* [10], on top of the Curve25519 elliptic curve. We use an existing blind signature implementation [1] based on RSA with 2048-bit keys and 1536-bit hashes. We use Ed25519 for digital signatures.

As discussed in §3, Ghostor uses external systems for anonymous communication and payment. In our implementation, clients use Tor [29] to communicate with the server and Zcash 1.0.15 for anonymous payments. We build a Zcash test network, separate from the Zcash main network. Ghostor,

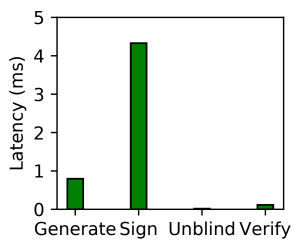
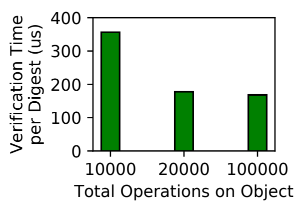


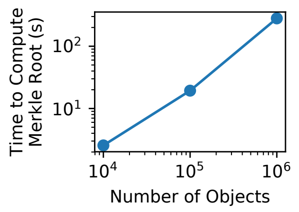
Figure 5: Blind signature

A	50% R, 50% W
B	95% R, 5% W
C	100% R
D	95% R, 5% Insert
E	95% R, 5% Range
F	50% R, 50% R-Modify-W

Figure 6: YCSB workloads (R: read, W: write)



(a) Run verification procedure



(b) Compute Merkle root

Figure 7: Operations for verification

however, could also be deployed on the Zcash main chain. Zcash is also used as the blockchain to post checkpoints. Our implementation runs as a *single* Ghostor server that stores its data in a scalable, fault-tolerant, distributed storage cluster. We discuss how to scale to *multiple* servers in Appendix B.

We implemented a proof of concept of our theoretical scheme Ghostor-MH (§7.2), in ≈ 2100 additional LOC. As it is a theoretical scheme, our focus in evaluating Ghostor-MH is simply to understand the latency of operations. Ghostor-MH includes AnonRAM's functionality, which, to our knowledge, has not been previously implemented. We omit zero-knowledge proofs in our implementation, as they are similar to AnonRAM and are not Ghostor-MH's innovation.

9 Evaluation

We run our experiments on Amazon EC2. Ghostor's storage cluster consists of three `i3en.xlarge` servers. We configure Ceph to replicate each object (key-value pair) on two SSDs on different machines, for fault-tolerance.

9.1 Microbenchmarks

Basic Crypto Primitives. We measured the latency of crypto operations used in Ghostor's critical path. En/decryption of object contents varies linearly with the object size, and takes ≈ 2 ms for 1 MiB. Key-private en/decryption for object headers and signing/verification of digests takes less than 150 us.

Blind Signatures. We also measure the blind signature scheme used for object creation, which consists of four steps. (1) The client *generates* a blinded hash of a random number. (2) The server *signs* the blinded hash. (3) The client *unblinds* the signature, obtaining the server's signature over the original number. (4) The server *verifies* the signature and the number during object creation. Results are shown in Fig. 5.

Verification Procedure. In Fig. 7, we measure the overhead of verification for digests in a single epoch. For client verification time, we perform an end-to-end test, measuring the total time to fetch digests and to verify them. The client has 1,000 signed digests for operations the client performed during the epoch that the client needs to check were included in the history of digests. We vary the total number of digests in the object's history for that epoch. The reported values in Fig. 7a are the total time to verify the object, divided by the total number of operations on the object, indicating the verification time *per digest*. The trend indicates a constant overhead when the total number of operations on the object is small, that is amortized when the number of operations is large.

Fig. 7b shows the server's overhead to compute the Merkle root. We inserted objects using YCSB (§9.2.2) during an epoch, and measured the time to compute the Merkle root at the end of that epoch. For 10,000 objects, this takes about 2.5 seconds; for 1,000,000 objects, it takes about 280 seconds. Reading the latest digest for each object (leaves of the Merkle tree) dominates the time to compute the Merkle root (2 seconds for 10,000 objects, 272 seconds for 1,000,000 objects). The reason is that our on-disk data structures are optimized for single-object operations, which are in the critical path. In particular, each object's digest chain is stored as a separate batched linked list, so reading the latest digests requires a separate read for each object.

9.2 Server-Side Overhead

This section measures to what extent anonymity and VerLinear affect Ghostor's performance. To ensure that the bottleneck was on the server, we set proof of work to minimum difficulty and do not use anonymous communication (§3), but we return to evaluating these in §9.3.

We measure the end-to-end performance of operations in Ghostor, both as a whole and for instantiations of Ghostor having only anonymity or VerLinear. We compare these to an insecure baseline as well as to competitive solutions for privacy and verifiable consistency, as we now describe.

1. Insecure system ("Insec"). This system uses the traditional ACL-based approach for serving objects. Each object access is preceded by a read to the object's ACL to verify that the user has permission to access the object. Similarly, creating an object requires a read to a per-user account file. It provides no security against a compromised server.

2. End-to-End Encrypted system ("E2EE"). This system encrypts objects placed on the server using end-to-end encryption similarly to SiRiUS [35]. Such systems have an encrypted KeyList similar to Ghostor's, but clients can cache their keys locally on most accesses unlike Ghostor.

3. Ghostor's anonymity system ("Anon"). This is Ghostor with VerLinear disabled. This fits a scenario where one wants to hide information from a *passive* server attacker. Unlike the E2EE system above, this system cannot cache keys locally—every operation incurs an additional round trip to fetch the KeyList from the server. In addition, every operation incurs yet another round trip at the beginning for the client to perform a proof of work. On the positive side, the server does not maintain any per-user ACL.

4. Fork Consistent system ("ForkC"). This system maintains Ghostor's digest chain (§5.1), but does not post checkpoints. Each operation appends to a per-object log of digests, using the techniques in §5.4. This system also performs an ACL check when creating an object.

5. Ghostor's VerLinear system ("VLinear"). This system corresponds to the VerLinear mechanism in §5 (including §5.2). This matches a use case where one wants integrity, but does not care about privacy. We do not include the verification procedure, already evaluated in §9.1.

6. Ghostor. This system achieves both anonymity and VerLinear, and therefore incurs the costs of both guarantees.

9.2.1 Object Accesses

In each setup, we measured the latency for create, `GET`, and `PUT` operations (Fig. 8a), throughput for `GET`s/`PUT`s to a single object (Fig. 9a), and the throughput for creating objects and for `GET`s/`PUT`s to multiple objects (Fig. 9b).

Fork consistency adds substantial overhead, because additional accesses to persistent storage are required for each operation, to maintain each object's log of digests. Ghostor, which both maintains a per-object log of digests and provides anonymity, incurs additional overhead because clients do not cache keys, requiring the server to fetch the header for each operation. In contrast, for Anon, the additional cost of reading the header is offset by the lack of ACL check. For 1 MiB objects, en/decryption adds a visible overhead to latency.

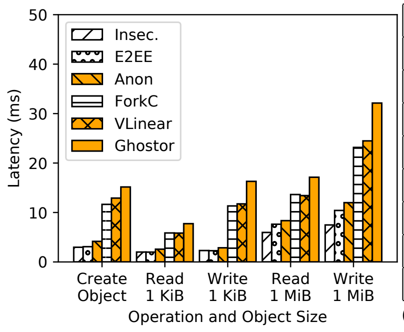
End-to-end encryption adds little overhead to throughput; this is because we are measuring throughput at the *server*, whereas encryption and decryption are performed by *clients*. The only factor affecting server performance is that the ciphertexts are 40 bytes larger than plaintexts.

Single-object throughput is lower for ForkC, VLinear, and Ghostor, because maintaining a digest chain requires requests to be serialized across multiple accesses to persistent storage. In contrast, Insec, E2EE, and Anon serve requests in parallel, relying on Ceph's internal concurrency control.

In the multi-object experiments, in which no two concurrent requests operate on the same object, this bottleneck disappears. For small objects, throughput drops in approximately an inverse pattern to the latency, as expected. For large objects, however, all systems perform commensurately. This is likely because reading/writing the object itself dominated the throughput usage for these experiments, without any concurrency overhead at the object level to differentiate the setups.

9.2.2 Yahoo! Cloud Serving Benchmark

In this section, we evaluate our system using the Yahoo! Cloud Serving Benchmark (YCSB). YCSB provides different workloads representative of various use cases, summarized in Table 6. We do not use Workload E because it involves range queries, which Ghostor does not support. As shown in Fig. 9c, anonymity incurs up to a 25% overhead for benchmarks containing insertions, owing to the additional accesses to storage required to store used object creation tokens. However, it shows essentially no overhead for `GET`s and `PUT`s. Fork



(a) Latency benchmarks

Operation	ms
Proof of Work	0.57
Read Header	1.1
Cl. Processing	0.68
Check Cl. Digest	0.14
Read/Fill Digest	3.2
Append Digest	1.5
Read Data	2.1
Cl. Processing	9.1

(b) Latency Breakdown for Ghostor, Read 1 MiB

Figure 8: Latency measurements

consistency adds a 3–4x overhead compared to the Insec baseline. VerLinear adds essentially no overhead on top of fork consistency; this is to be expected, because the overhead of VerLinear is outside of the critical path (except for insertions, where the overhead is easily amortized). Ghostor, which provides both anonymity and VerLinear, must forgo client-side caching, and therefore incurs additional overhead, with a 4–5x throughput reduction overall compared to the Insec baseline.

9.3 End-to-End Latency

We now analyze the performance of Ghostor from the client's perspective, including the cost of proof of work and anonymous communication (§3).

9.3.1 Microbenchmarks

The latency experienced by a Ghostor client is the latency measured in Fig. 8, plus the additional overhead due to the proof of work mechanism and anonymous communication. The difficulty of the proof of work problem is adjustable. For the purpose of evaluation, we set it to a realistic value to prevent denial of service. Fig. 8b indicates that it takes ≈ 32 ms for a Ghostor operation; therefore, we set the proof of work difficulty such that it takes the client, on average, 100 times longer to solve (≈ 3.2 s). Fig. 10 shows the distribution of latency for the client to solve the proof of work problem. As expected, the distribution appears to be memoryless.

In our implementation, a client connects to a Ghostor server by establishing a circuit through the Tor [29] network. The performance of the connection, in terms of both latency and throughput, varies according to the circuit used. Fig. 10 shows the distribution of (1) circuit establishment time, (2) round-trip time, and (3) network bandwidth. We used a fresh Tor circuit for each measurement. Based on our measurements, a Tor circuit usually provides a round-trip time less than 1 second and bandwidth of at least 2 Mb/s.

9.3.2 Macrobenchmarks

We now measure the end-to-end latency of each operation in Ghostor's client API (§2), including all overheads experienced by the client. As explained in §9.3.1, the overhead due to proof of work and Tor is quite variable; therefore, we repeat each experiment 1000 times, using a separate Tor circuit each time, and report the distribution of latencies for each operation in Fig. 12. Comparing Fig. 12 to Fig. 8, the client-side latency is dominated by the cost of PoW and Tor; Ghostor's core techniques in Fig. 8 have relatively small latency overhead. For the `pay` operation, we measure only the time to redeem a Zcash payment for a single token, not the time for proof of work or making the Zcash payment (see §9.4 for a discussion of this overhead). `GET` and `PUT` for large objects are the slowest, because Tor network bandwidth becomes a bottleneck. The `create_user` operation (not shown in Fig. 12) is only 132 microseconds, because it generates an El Gamal keypair locally without any interaction with the server.

9.4 Zcash

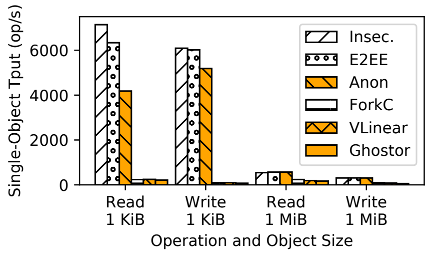
In our implementation, we build our own Zcash test network to avoid the expense from Zcash's main network. Since our system leverages Zcash in a minimal way, the overhead of Zcash is not on the critical path of our protocol. According to the Zcash website [105] and block explorer [2], the block size limit is about 2 MiB, and block interval is about 2.5 minutes. In the past six months, the maximum block size has been less than 150 KiB and the average transaction fee has been much less than 0.001 ZEC (0.05 USD at the time of writing). Hence, even with shorter epochs (less time for misbehavior detection), the price of Ghostor's checkpoints is modest since there is a single checkpoint per epoch for the whole system.

9.5 Ghostor-MH

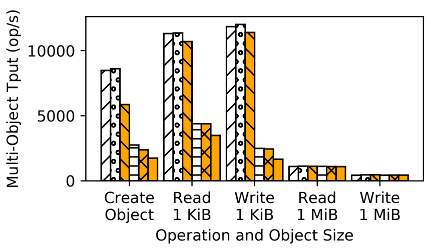
For completeness, we evaluate the *theoretical* Ghostor-MH scheme presented in §7.2, focusing only on the latency of accessing an object. We do not use Tor and we set the PoW difficulty to minimum. Latency is dominated by en/decryption on the client, because object contents and ORAM state are encrypted with El Gamal encryption, which is much slower than symmetric-key encryption. Fig. 11a shows the object access latency for an object group, as we vary its size. It scales logarithmically, as expected from Path ORAM. An additional overhead of ≈ 2 s comes from re-encrypting ORAM client state (32 KiB, after padding and encryption) on each access. Fig. 11b shows the object access latency as we vary the number of object groups (each object group is 31 KiB). It scales linearly, because the client makes fake accesses to *all* other object groups to hide which one it truly accessed. Latency could be improved by using multiple client CPU cores.

10 Related Work

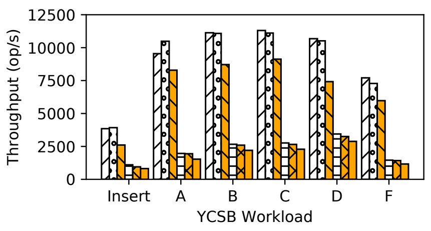
Systems Providing Consistency. We have already compared extensively with SUNDR [64]. Venus [87] achieves eventual consistency; however, Venus requires some clients to be frequently online and is vulnerable to malicious clients. Caelus [55] has a similar requirement and does not resist collusion of malicious clients and the server. Verena [49] trusts one of two servers. SPORC [31], which combines fork consistency with operational transformation, allows clients to recover from a fork attack, but does not resist faulty clients. Depot [67] can tolerate faulty clients, but achieves a weaker



(a) Single-Object Throughput



(b) Multi-Object Throughput



(c) Throughput for YCSB

Figure 9: Benchmarks comparing throughput of the six setups described in §9.2

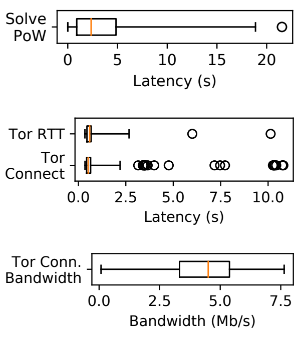
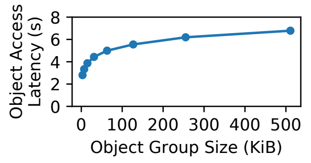
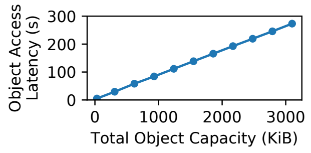


Figure 10: Microbenchmarks of PoW mechanism and Tor



(a) Latency vs. Object Group Size



(b) Latency vs. No. Object Groups

Figure 11: Ghostor-MH

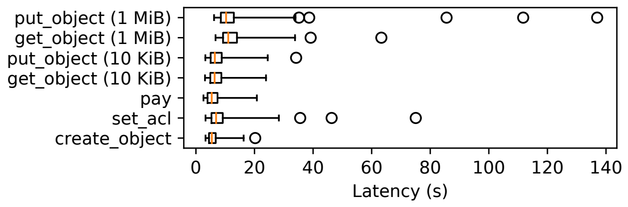


Figure 12: End-to-end latencies of client-side operations

notion of consistency than VerLinear. Furthermore, its consistency techniques are at odds with anonymity. Ghostor and these systems use hash chains [39, 68] as a key building block.

Systems Providing E2EE. Many systems provide end-to-end encryption (E2EE), but leak significant user information as discussed in §3.3: academic systems such as Persona [8], DEPSKY [13], CFS [14], SiRiUS [35], Plutus [48], ShadowCrypt [41], M-Aegis [60], Mylar [75] and Sieve [99] or industrial systems such as Crypho [27], Tresorit [46], Keybase [52], PreVeil [76], Privly [77] and Virtru [98].

Systems Using Trusted Hardware. Some systems, such as Haven [9] and A-SKY [25], protect against a malicious server by using trusted hardware. Existing trusted hardware, like Intel SGX, however, suffer from side-channel attacks [96].

Oblivious Systems. A complementary line of work to Ghostor aims to hide access patterns: *which* object was accessed. Standard Oblivious RAM (ORAM) [36, 86, 100] works in the single-client setting. Multi-client ORAM [7, 40, 50, 65, 66, 80, 90] extends ORAM to support multiple clients. These works either rely on central trust [80, 90] (either a fully trusted proxy or fully trusted clients) or provide limited functionality (not providing global object *sharing* [7], or revealing user identities [66]). GORAM [65] assumes the adversary controlling the server does not collude with clients. Furthermore, it only provides obliviousness within a single data owner's objects, not *global obliviousness* across all data owners.

AnonRAM [7] and PANDA [40] provide global obliviousness and hide user identity, but are slow. They do not provide for sharing objects or mitigate resource abuse. One can realize these features by applying Ghostor's techniques to these schemes, as we did in §7.2 to build Ghostor-MH. Unlike these schemes, Ghostor-MH is a *metadata-hiding object-sharing scheme* providing both global obliviousness and anonymity without trusted parties or non-collusion assumptions.

Decentralized Storage. Peer-to-peer storage systems, like OceanStore [56], Pastry [79], CAN [78], and IPFS [12], allow users to store objects on globally distributed, untrusted storage without any coordinating central trusted party. These systems are vulnerable to rollback/fork attacks on mutable data by malicious storage nodes (unlike Ghostor's VerLinear). While some of them encrypt objects for privacy, they do not provide a mechanism to distribute secret keys while preserving anonymity, as Ghostor does. Recent blockchain-based decentralized storage systems, like Storj [92], Swarm [94], Filecoin [32], and Sia [88], have similar shortcomings.

Decentralized Trust. As discussed in §1, blockchain systems [17, 20, 73, 103] and verifiable ledgers [61, 71] can serve as the source of decentralized trust in Ghostor.

Another line of work aims to provide efficient auditing mechanisms. EthIKS [15] leverages smart contracts [17] to monitor key transparency systems [71]. Catena [93] builds log systems based on Bitcoin transactions, which enables efficient auditing by low-power clients. It may be possible to apply techniques from those works to optimize our verification procedure in §5.2. However, none of them aim to build secure data-sharing systems like Ghostor.

Secure Messaging. Secure messaging systems [26, 95, 97] hide network traffic patterns, but they do not support object storage/sharing as in our setting. Ghostor can complementarily use them for its anonymous communication network.

11 Conclusion

Ghostor is a data-sharing system that provides *anonymity* and *verifiable linearizability* in a strong threat model that assumes only *decentralized trust*.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Carmela Troncoso, for their invaluable feedback. We would also like to thank students from the RISELab Security Group and BETS Research Group for giving us feedback on early drafts, and David Culler for advice and discussion.

This work has been supported by NSF CISE Expeditions Award CCF-1730628, as well as gifts from the Sloan Foundation, Bakar, Alibaba, Amazon Web Services, Ant Financial, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. This research is also supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1752814. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] https://github.com/cryptoballot/rsablind.

[2] BitInfoCharts. https://bitinfocharts.com/zcash/.

[3] Open mHealth. http://www.openmhealth.org/. Sep. 19, 2019.

[4] S. G. Akl and P. D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *TOCS*, 1983.

[5] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman. Medrec: Using blockchain for medical data access and permission management. In *OBD*, 2016.

[6] A. Back. Hashcash - a denial of service countermeasure. 2002.

[7] M. Backes, A. Herzberg, A. Kate, and I. Pryvalov. Anonymous RAM. In *ESORICS*, 2016.

[8] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: an online social network with user-defined privacy. In *CCR*, 2009.

[9] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *TOCS*, 2015.

[10] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval. Key-privacy in public-key encryption. In *ASIACRYPT*, 2001.

[11] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *S&P*, 2014.

[12] J. Benet. IPFS: Content addressed, versioned, P2P file system. *CoRR*, 2014.

[13] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *TOS*, 2013.

[14] M. Blaze. A cryptographic file system for UNIX. In *CCS*, 1993.

[15] J. Bonneau. EthIKS: Using Ethereum to audit a CONIKS key transparency log. In *FC*, 2016.

[16] F. Buccafurri, G. Lax, S. Nicolazzo, and A. Nocera. Accountability-preserving anonymous delivery of cloud services. In *TrustBus*, 2015.

[17] V. Buterin et al. Ethereum white paper. *GitHub repository*, 2013.

[18] J. L. Camenisch, J. Piveteau, and M. A. Stadler. Blind signatures based on the discrete logarithm problem. In *EUROCRYPT*, 1994.

[19] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[20] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[21] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *CACM*, 1981.

[22] D. Chaum. Blind signatures for untraceable payments. In *EUROCRYPT*, 1983.

[23] D. Chaum. Blind signature system. In *EUROCRYPT*, 1984.

[24] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *S&P*, 2010.

[25] S. Contiu, S. Vaucher, R. Pires, M. Pasin, P. Felber, and L. Réveillère. Anonymous and confidential file sharing over untrusted clouds. *SRDS*, 2019.

[26] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *S&P*, 2015.

[27] Crypho. Enterprise communications with end-to-end encryption. https://www.crypho.com/.

[28] J. Davis. The 10 biggest healthcare data breaches of 2019, so far. https://healthitsecurity.com/news/the-10-biggest-healthcare-data-breaches-of-2019-so-far. Sep. 12, 2019.

[29] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. Technical report, 2004.

[30] A. Eijdenberg, B. Laurie, and A. Cutter. Verifiable data structures. https://github.com/google/trillian/blob/master/docs/VerifiableDataStructures.pdf.

[31] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *OSDI*, 2010.

[32] Filecoin. https://filecoin.io. Apr. 16, 2019.

[33] W. C. Garrison, A. Shull, S. Myers, and A. J. Lee. On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *S&P*, 2016.

[34] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 2002.

[35] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*, 2003.

[36] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 1996.

[37] W. Gordon, A. Chopra, and A. Landman. Patient-led data sharing — a new paradigm for electronic health data. https://catalyst.nejm.org/patient-led-health-data-paradigm/. Sep. 12, 2019.

[38] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, 2016.

[39] S. Haber and W. S. Stornetta. How to time-stamp a digital document. In *EUROCRYPT*, 1990.

[40] A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs. Private anonymous data access. 2019.

[41] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. Shadowcrypt: Encrypted web applications for everyone. In *CCS*, 2014.

[42] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.

[43] D. Hoppe. Blockchain use cases: Electronic health records. https://gammalaw.com/blockchain_use_cases_electronic_health_records/. Sep. 12, 2019.

[44] R. Hurst and G. Belvin. Security through transparency. https://security.googleblog.com/2017/01/security-through-transparency.html.

[45] Identity Theft Resource Center. At mid-year, U.S. data breaches increase at record pace. In *ITRC*, 2018.

[46] Tresorit Inc. End-to-end encrypted cloud storage. tresorit.com.

[47] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[48] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, 2003.

[49] N. Karapanos, A. Filios, R. A. Popa, and S. Capkun. Verena: End-to-end integrity protection for web applications. In *S&P*, 2016.

[50] N. P. Karvelas, A. Peter, and S. Katzenbeisser. Using oblivious RAM in genomic studies. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. 2017.

[51] B. Kepes. Some scary (for some) statistics around file sharing usage, 2015. https://www.computerworld.com/article/2991924/some-scary-for-some-statistics-around-file-sharing-usage.html.

[52] Keybase.io. https://keybase.io/.

[53] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais. Commit-chains: Secure, scalable off-chain payments, 2018. https://eprint.iacr.org/2018/642.

[54] S. M. Khan and K. W. Hamlen. AnonymousCloud: A data ownership privacy provider framework in cloud computing. In *TrustCom*, 2012.

[55] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *S&P*, 2015.

[56] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[57] S. Kumar, Y. Hu, M. P Andersen, R. A. Popa, and D. E. Culler. JEDI: Many-to-many end-to-end encryption and key delegation for IoT. In *USENIX Security*, 2019.

[58] L. Lamport. The part-time parliament. *TOCS*, 1998.

[59] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 2001.

[60] B. Lau, S. P. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva. Mimesis Aegis: A mimicry privacy shield-a system's approach to data privacy on public cloud. In *USENIX Security*, 2014.

[61] B. Laurie, A. Langley, and E. Kasper. Certificate transparency. Technical report, 2013.

[62] R. Lemos. Home Depot estimates data on 56 million cards stolen by cybercriminals. https://arstechnica.com/information-technology/2014/09/home-depot-estimates-data-on-56-million-cards-stolen-by-cybercrimnals/. Apr. 21, 2019.

[63] H. M. Levy. *Capability-based computer systems*. 1984.

[64] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[65] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. Privacy and access control for outsourced personal records. In *S&P*, 2015.

[66] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. Maliciously secure multi-client ORAM. In *ACNS*, 2017.

[67] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *TOCS*, 2011.

[68] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *USENIX Security*, 2002.

[69] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *PODC*, 2002.

[70] Medicalchain - blockchain for electronic health records. https://medicalchain.com. Sep. 12, 2019.

[71] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: bringing key transparency to end users. In *USENIX Security*, 2015.

[72] A. Mettler, D. A. Wagner, and T. Close. Joe-E: A security-oriented subset of java. In *NDSS*, 2010.

[73] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[74] V. Pacheco and R. Puttini. SaaS anonymous cloud service consumption structure. In *ICDCS*, 2012.

[75] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *NSDI*, 2014.

[76] PreVeil Inc. PreVeil: End-to-end encryption for everyone. preveil.com.

[77] Privly Inc. Privly. priv.ly.

[78] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[79] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[80] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *S&P*, 2016.

[81] T. Seals. 17% of workers fall for social engineering attacks, 2018.

[82] Secret Double Octopus | passwordless high assurance authentication. https://doubleoctopus.com. Apr. 21, 2019.

[83] A. Shamir. How to share a secret. *CACM*, 1979.

[84] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *SOSP*, 1999.

[85] J. Sharp. Will healthcare see ethical patient data exchange? https://www.idigitalhealth.com/news/healthcare-ethical-patient-data-exchange-cms-rule. Sep. 12, 2019.

[86] E. Shi, T. H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[87] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *CCSW*, 2010.

[88] Sia. https://sia.tech. Apr. 16, 2019.

[89] M. Srivatsa and M. Hicks. Deanonymizing mobility traces: Using social network as a side-channel. In *CCS*, 2012.

[90] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *S&P*, 2013.

[91] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple Oblivious RAM protocol. In *CCS*, 2013.

[92] Decentralized cloud storage — Storj. https://storj.io. Apr. 16, 2019.

[93] A. Tomescu and S. Devadas. Catena: Efficient non-equivocation via Bitcoin. In *S&P*, 2017.

[94] V. Tron, A. Fischer, and N. Johnson. Smash-proof: Auditable storage for Swarm secured by masked audit secret hash. Technical report, Ethersphere, 2016.

[95] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. Stadium: A distributed metadata-private messaging system. In *SOSP*, 2017.

[96] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[97] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *SOSP*, 2015.

[98] Virtru Inc. Virtru: Email encryption and data protection solutions. www.virtru.com.

[99] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *NSDI*, 2016.

[100] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS*, 2015.

[101] S. A Weil, S. A. Brandt, E. L. Miller, D. DE Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[102] WhatsApp. WhatsApp's privacy notice. www.whatsapp.com/legal/?doc=privacy-policy, 2012.

[103] M. Yin, D. Malkhi, M. Reiterand, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *PODC*, 2019.

[104] S. Zarandioon, D. D. Yao, and V. Ganapathy. K2C: Cryptographic cloud storage with lazy revocation and anonymous access. In *International Conference on Security and Privacy in Communication Systems*, 2011.

[105] Zcash. Zcash: All coins are created equal. http://z.cash/.

[106] K. Zetter. 'Google' hackers had ability to alter source code. https://www.wired.com/2010/03/source-code-hacks/. Apr. 21, 2019.

[107] K. Zetter. An unprecedented look at Stuxnet, the world's first digital weapon. https://www.wired.com/2014/11/countdown-to-zero-day-stuxnet/. Apr. 21, 2019.

A Full Protocol Description for Ghostor

Below, we describe the client-server protocol used by Ghostor.

A.1 `GET` Protocol

1. Server sends a PoW challenge to the client (§6).
2. Client sends the server the PoW solution, PVK of the object that the user wishes to access, and the server returns the object header and current epoch.
3. The client assembles a digest for the `GET` operation, including the epoch number, PVK, RVK, WVK, and a random nonce, and signs it with RSK (obtained from the header). It sends the signed digest to the server.
4. Server reads the latest digest and checks that the client's candidate digest is consistent with it. If not (for example, if the header was changed in-between round trips), the server gives the client the object header, and the protocol returns to Step 3.
5. Server adds $\text{Hash}_{\text{prev}}$, $\text{Hash}_{\text{header}}$, and $\text{Hash}_{\text{data}}$ to the digest (according to the order in which it commits operations on the object). Then it signs it and adds it to the log of digests for that object.
6. Server returns the object contents and the digest, including the server's signature, to the client.
7. Client checks that the signed digest matches the object contents and digest that the client provided. If so, it returns the object contents to the user and sends the signed digest to the verification daemon.

A.2 `PUT` Protocol

1. Server sends a PoW challenge to the client (§6).
2. Client sends the server the PoW solution and PVK of the object to `PUT`, and the server returns the object header, current epoch, and latest server-signed digest for that object.
3. The client assembles a PREPARE digest for the write operation, including the epoch number, PVK, RVK, WVK, and signs it with WSK (obtained from the header). It sends the signed digest to the server.
4. Server reads the latest digest and checks that the client's candidate digest is consistent with it. If not, then the server gives the client the object header, and the protocol returns to Step 3.
5. Server adds $\text{Hash}_{\text{prev}}$, $\text{Hash}_{\text{header}}$, and $\text{Hash}_{\text{data}}$ to the digest (according to the order in which it commits operations on the object). Then it signs it and adds it to the log of digests for that object.
6. Server returns the signed digest to the client.
7. Client assembles a COMMIT digest for the write operation, including the same fields as the PREPARE digest, and also $\text{Hash}_{\text{prep}}$ and $\text{Hash}_{\text{data}}$ according to the new data. Then it signs it and uploads it to the server, including the new object contents.
8. Server decides if this `PUT` "wins." It wins as long as no other `PUT` whose PREPARE digest is after this `PUT`'s PREPARE digest has already committed. If this `PUT` wins, then the server performs the write, signs the digest, and adds it

to the log of digests for that object. If not, it still signs the digest and adds it to the log, but it replaces $\mathsf{Hash}_{\mathsf{data}}$ with the current hash of the data, including the value provided by the client as an "addendum" so that the verification daemon can still verify the client's signature. The server may also reject the COMMIT digest if the key list changed meanwhile due to a **set_acl** operation.

9. Server returns the digest, including the server's signature, to the client.
10. Client checks that the signed digest matches the object contents and digest that the client provided. If so, it sends the signed digest to the verification daemon.

A.3 Access Control

1. Server sends a PoW challenge to the client (§6).
2. Client sends the server the PoW solution and PVK of the object to write, and the server returns the object header, current epoch, and latest server-signed digest for that object.
3. The client assembles a digest for the write operation, including all fields, and signs it with PSK. It sends the signed digest to the server. Client also signs PVK with PSK and includes that signature in the request. Client also includes the new header.
4. Server acquires a lock (lease) on the object for this client (unless it is already held for this client), reads the latest digest, and checks that the client's candidate digest is consistent with it. If not, then the server gives the client the object header, and the protocol returns to Step 3. When returning to Step 3, the server checks if the client's signature over PVK is correct. If so, the server holds the lock on the object during the round trip. If not, the server releases it.
5. Server updates the header, signs the digest, adds it to the log of digests for that object, and releases the lock.
6. Server returns the digest, including the server's signature, to the client.
7. Client checks that the signed digest matches the object contents and digest that the client provided. If so, it returns the object contents to the user and sends the signed digest to the verification daemon.

The owner of the object generates new keys and encrypts the object under the new key. If a user is being granted access, the owner may still generate new keys to prevent the server from learning whether or not a user was revoked. The owner shuffles the key list upon any change to it. The owner may also add padding to hide the number of users in the key list.

A.4 Object Creation

1. Server sends a PoW challenge to the client (§6)
2. Client sends the server the PoW, PVK of the object that the user wishes to create, a token signed by the server for proof of payment (§2), the header for the new object, and the object's first digest (for which $\mathsf{Hash}_{\mathsf{prev}}$ is empty). This involves generating all the keys in Fig. 4) for the new object.
3. Server verifies the signature on the token, and checks that it has not been used before.
4. Server "remembers" the hash of the token by storing it in permanent storage.
5. Server writes the object header. It signs the digest and creates a log for this object containing only that digest.
6. Server returns the digest, including the server's signature, to the client.
7. Client checks that the signed digest matches the object contents and digest that the client provided. If so, it returns the object contents to the user and sends the signed digest to the verification daemon.

A.5 Verification Procedure

At the end of each epoch, the verification daemon downloads the digest chain and checkpoints to verify operations performed in the epoch.

1. Server sends a PoW challenge to the daemon (§6). (The server will request additional PoWs for long lists of digests as it streams them to the daemon in Step 3.)
2. Daemon responds with PoW and requests the object's digest chain from the server for that epoch. It sends the server a signed digest for that object, so the server knows this is a legitimate request.
3. Server returns the digest chain for that object, along with a Merkle proof.
4. Daemon retrieves the Merkle root from the checkpoint in Zcash, and verifies the server's Merkle proof to check that the last digest in the digest chain is included in the Merkle tree at the correct position based on the object's PVK.
5. Daemon verifies that all digests corresponding to the user's operations are in the digest chain, and that the diges chain is valid.

To check that the digest chain is valid, the daemon checks:

1. $\mathsf{Hash}_{\mathsf{prev}}$ for each digest matches the previous digest. If this digest is the first digest in this epoch, the previous digest is the last digest in the previous epoch. The daemon knows this previous digest already since the daemon must have checked the previous epoch. If this is the first epoch, then $\mathsf{Hash}_{\mathsf{prev}}$ should be empty.
2. $\mathsf{Hash}_{\mathsf{prep}}$ in each COMMIT digests matches an earlier PREPARE digest in the same epoch, and each PREPARE digest matches with at most one COMMIT digest.
3. $\mathsf{Hash}_{\mathsf{data}}$ only changes in winning COMMIT digests, which are signed with WSK.
4. WVK, RVK, and $\mathsf{Hash}_{\mathsf{keylist}}$ only change in digests signed with PSK, and PVK never changes.
5. The epoch number in digests matches the epoch that the client requested, and never decreases from one digest to the next.
6. $\mathsf{Sig}_{\mathsf{client}}$ is valid and signed using the correct signing key. For example, if this operation is read, $\mathsf{Sig}_{\mathsf{client}}$ must be signed using RSK.

A.6 Payment

First, the user pays the server using an anonymous cryptocurrency such as Zcash [105], and obtains a proof of payment from Zcash. Then, the client obtains tokens from the server, as follows:

1. Server sends a PoW challenge to the client (§6).
2. Client sends the server the PoW, proof of payment, and *t* blinded tokens, where *t* corresponds to the amount paid.
3. Server checks that the proof of payment is valid and has not been used before.
4. Server "remembers" the proof of payment by storing it in persistent storage.
5. Server signs the blinded tokens, ensuring that *t* indeed corresponds to the amount paid, and sends the signed blinded tokens to the client.
6. Client unblinds the signed tokens and saves them for later use.

B Extension: Scalability

Our implementation of Ghostor that we evaluated in §9 consists of a single Ghostor server, which stores data in a storage cluster that is internally replicated and fault-tolerant (Ceph RADOS). In this appendix, we discuss techniques to scale this setup by replicating the Ghostor server as well.

Given that we consider a malicious adversary, it may seem natural to use PBFT [20]. PBFT, however, is neither necessary nor sufficient in Ghostor's setting. It is not necessary because we already post checkpoints to a ledger based on decentralized trust (§5.2) to achieve verifiable integrity. It is not sufficient because we assume an adversary who can compromise any few machines across which we replicate Ghostor, which is incompatible with Byzantine Fault Tolerance.

The primary challenge to replicating the Ghostor server is *synchronization*: if multiple operations on the same object may be handled by different servers, the servers may concurrently mutate the on-disk data structure for that object. A simple solution is to use object-level locks provided by Ceph RADOS. This is probably sufficient for most uses. But, if server-side caching of objects in memory is implemented, caches in the Ghostor servers would have to be kept coherent.

Alternatively, one could partition the object space among the servers, so each object has a single server responsible for processing operations on it. A set of *load balancer* servers run Paxos [58, 59] to arrive at a consensus on which servers are up and running, so that requests meant for one server can be re-routed to another if it goes down. Note that Paxos is outside of the critical path; it only reacts to failures, not to individual operations. Based on the consensus, the load balancers determine which server is responsible for each object. Because all objects are stored in the same storage pool, the objects themselves do not need to be moved when Ghostor servers are added or removed, only when storage servers are added or removed (which is handled by Ceph). Object-level locks in Ceph RADOS would still be useful to enforce that at most one server is operating on a Ghostor object at a time.

C Extension: Files and Directories

Our design of Ghostor can be extended to support a hierarchy of directories and files. Each directory or file corresponds to a PVK and associated Ghostor object; the PVK has a similar role to an inode number in a traditional file system. The Ghostor object corresponding to a directory contains a mapping from name to PVK as a list of *directory entries*. Given the PVK of a root directory and a filepath, a client iteratively finds the PVK of each directory from left to right; in the end, it will have the PVK of the file, allowing it to access the Ghostor object corresponding to a file. The procedure is analogous to resolving a filepath to an inode number in a traditional file system. The Ghostor object corresponding to a file may either contain the file contents directly, or it may contain the PVKs of other objects containing the file data, like an inode in a traditional file system.

The "no user-side caching" principle §4 applies here, in the sense that clients may not cache the PVK of a file after resolving it once. A client must re-resolve a file's PVK on each access; caching the PVK and accessing the file without first accessing all parent directories would reveal that the same user has accessed the file before.

D Additional Description of Ghostor-MH

§7.2 explains Ghostor-MH at a high level. §8 and §9 describe our implementation and evaluation of Ghostor-MH.

Appendix D.2 below provides a more in-depth explanation of Ghostor-MH. We first provide more details about AnonRAM in Appendix D.1. This is necessary because, as explained in §7.2, we construct Ghostor-MH by applying Ghostor's techniques to AnonRAM [7].

D.1 Overview of AnonRAM

ORAM [36] is a technique to access objects on a remote server without revealing which objects are accessed. Many ORAM schemes, such as Path ORAM [91], allow a *single user* to access data. Path ORAM [91] works by having the client shuffle a small amount of server-side data with each access, such that the server cannot link requests to the same object. Clients store mutable *secret state*, including a stash and position map, used to find objects after shuffling.

AnonRAM extends single-user ORAM to support *multiple users*. Each AnonRAM user essentially has her own ORAM on the server. When a user accesses an object, she (1) performs the access as normal in her own ORAM, and (2) performs a *fake access* to all of the other users' ORAMs. To the server, the fake accesses are indistinguishable from genuine accesses, so the server does not learn to which ORAM the user's object belongs. This, together with each individual ORAM hiding which of its objects was accessed, results in global obliviousness across all objects in all ORAMs.

To support fake accesses, *re-randomizable* public-key encryption (e.g., El Gamal) is used to encrypt objects in each

ORAM. To guard against malicious clients, the server requires a zero-knowledge proof with each real or fake access, to prove that *either* (1) the client knows the secret key for the ORAM, *or* (2) the new ciphertexts encrypt the same data as existing ciphertexts (i.e., they were re-randomized correctly).

A limitation of AnonRAM is that there is *no object sharing among users*; each user can access only the objects she owns. Furthermore, AnonRAM and similar schemes (§10) are *theoretical*—they consider oblivious storage from a cryptographic standpoint, but do not consider challenges like payment, user accounts, and resource abuse.

D.2 Ghostor-MH

Recall from §7.2 that we apply to AnonRAM Ghostor's principle of switching from a user-centric to a data-centric design. Each ORAM now corresponds to an *object group*, which is a fixed-size set of objects with a shared ACL. Each object group has one object header and one digest chain.

Ghostor-MH uses Path ORAM, which organizes server-side storage as a binary tree. To guard against a malicious adversary controlling the server, we build a Merkle tree over the binary tree, and compute $\mathsf{Hash}_{\mathsf{data}}$ in each digest as the hash of the Merkle root and ORAM secret state. This allows each client to efficiently compute the new $\mathsf{Hash}_{\mathsf{data}}$ after each ORAM access, without downloading the entire ORAM tree. The ORAM secret state is stored on the server, encrypted with OSK, so multiple clients can access an object group. This is analogous to Ghostor's object header, which stores an object's keys encrypted on the server.

To access an object, a client (1) identifies the object group containing it, (2) downloads the object header and encrypted ORAM secret state, (3) obtains OSK from the object header, (4) decrypts the ORAM secret state, (5) uses it to perform the ORAM access, (6) encrypts and uploads the new ORAM secret state, (7) computes a new digest for the operation, (8) has the server sign it, and (9) sends it to the verification daemon. For all other object groups, the client performs a *fake access* that fetches data from the server and generates a digest, but only re-randomizes ciphertexts instead of performing a real access. This hides which object group contains the object. When writing an object, the client pads it to a maximum size (the ORAM block size) to hide the length of the object.

Below, we explain some more details about Ghostor-MH:

Fake accesses. OSK is replaced with an El Gamal keypair. This allows ciphertexts in the ORAM tree and the ORAM secret state to be re-randomized. We no longer attach a client signature to each digest, but instead modify the zero-knowledge proof in AnonRAM to prove that *either* the client can produce a signature over the digest with WSK, *or* the ciphertexts were properly re-randomized.

Hiding timing. Similar to secure messaging systems [97], Ghostor-MH operates in rounds (shorter than epochs) to hide timing. In each round, each client either accesses an object as described above, or performs a fake access on all ORAMs if there is no pending object access. Each client chooses a random time during the round to make its request to the server.

Using tokens. In a globally oblivious system like Ghostor-MH, it is impossible to enforce the per-object quotas discussed in §6.3. Thus, it is advisable to require users to expend tokens for *all* operations (except **pay**), not just **create_object**. Our PoW mechanism applies to Ghostor-MH unchanged.

Object group creation. The server can distinguish payment (to obtain tokens) and object group creation from `GET`/`PUT` operations. The most secure solution is to have a setup phase to create all object groups and perform all payment in advance. Barring this, we propose adding a special round at the start of each epoch, used only for creation and payment; all object accesses during an epoch happen after this special round.

List of object groups. To make fake accesses, each client must know the full list of object groups. To ensure this, we can add an additional digest chain to keep track of all created object groups, checkpointed every epoch with the rest of the system.

Changing permissions. In our solution so far, the server can distinguish a **set_acl** operation from object accesses. To fix this, we require the owner of each object group to perform exactly one **set_acl** for that object group during each epoch; if he does not wish to change it, he sets it to the same value.

Concurrency. When a client iterates over all ORAMs to make accesses (fake or real), the client locks each ORAM individually and releases it after the access. No "global lock" is held while a client makes fake accesses to all ORAMs.

E Ghostor's Privacy Guarantee

In this appendix, we use the simulation paradigm of Secure Multi-Party Computation (SMPC) [19] to define Ghostor's privacy guarantee. We begin in Appendix E.1 by providing an overview of our definition and proof sketch, along with an explanation of how our simulation-based definition matches the one in §3.3.

E.1 Overview

We formally define Ghostor's anonymity by specifying an *ideal world*. We provided a definition in §3.3, but we consider it to be informal because it does not clearly state what the adversary learns if some users are compromised/malicious. The ideal world is specified such that it is easy to reason about what information the adversary learns; what the adversary learns in the ideal world is our definition of what an anonymous object sharing system leaks to an adversary (i.e., what *anonymity* does not hide). In the ideal world, clients interact with an uncorruptible trusted party $\mathcal{F}$ called an *ideal functionality*. On each API call issued by a client, $\mathcal{F}$ services the request and provides to the adversary (denoted $\mathcal{S}$ in the ideal world) a well-defined subset of information in the API call. The subset of information that $\mathcal{F}$ gives to $\mathcal{S}$ defines what information Ghostor leaks to the adversary, and provides a clear definition of what anonymity means in our setting. To allow for a malicious adversary, $\mathcal{S}$ chooses what response is returned to the client. $\mathcal{S}$ may violate integrity in a way that the client will only detect at the end of the epoch (e.g., fork attack), but cannot deny service by returning a message that the client would immediately detect as fake (e.g., a message with a bad or missing signature).

To prove that Ghostor achieves that definition of anonymity, we additionally define a *real world*. The purpose of the real world is to model the Ghostor system in the abstract environment we used for the ideal world. In the real world, clients interact directly with the adversary (denoted $\mathcal{A}$ in the real world), which services the requests and learns some information. The protocol that clients use to interact with $\mathcal{A}$ is the same as that used in the actual Ghostor system.

In both worlds, there is another party $\mathcal{Z}$ called the environment. The environment can communicate freely with the adversary and decides what operations the clients issue.

E.1.1 Summary of Proof Sketch

To prove that Ghostor achieves our definition of anonymity as specified in the ideal world, we demonstrate that for every real-world adversary $\mathcal{A}$ in the real world, there exists an ideal-world adversary $\mathcal{S}$ in the ideal world such that the environment $\mathcal{Z}$ cannot distinguish whether it is interacting with the real world or the ideal world. Intuitively, this means that any "attack" that the real-world adversary $\mathcal{A}$ can perform in the real world, can also be performed by the ideal-world $\mathcal{S}$ in the ideal world. Because the ideal-world setup is, by definition, anonymous, this shows that any attacks that $\mathcal{A}$ can perform are those allowed by anonymity, which implies that the real-world setup achieves anonymity.

Given a real-world adversary $\mathcal{A}$, we construct the corresponding ideal-world adversary $\mathcal{S}$ via a simulation. This means that $\mathcal{S}$ uses $\mathcal{A}$ as a black box by carefully simulating a "real world" that runs in tandem with the ideal world.

E.1.2 Map to Definition of Anonymity in §3.3

In §3.3, we explained Ghostor's privacy guarantee in terms of a leakage function. Anonymity, as defined by our ideal world below, maps to the leakage function given in §3.3 as follows. The leakage function in §3.3 is largely the same as the information that $\mathcal{F}$ gives to $\mathcal{S}$ on each API call (Appendix E.2.2). There are a few minor differences, which we now explain. Timing information is not included in Appendix E.2.2 because the model we use in our cryptographic formalization does not have a notion of time. That said, the order in which the requests are processed is given to $\mathcal{S}$; it is implicit in the order in which $\mathcal{F}$ sends messages to it. Finally, although not explicit in Appendix E.2.2, $\mathcal{S}$ can infer how many round trips are performed between the client and server in processing each operation: as long as there is no client-side caching of data (§4.4), the adversary can infer how many round trips are required from the client-server protocol (Appendix A), because we do not model concurrently executing operations. We consider the protocol to be public, so this does not reveal any meaningful information.

Our definition of anonymity matches the everyday use of the word "anonymity" because $\mathcal{S}$ does not receive any user-specific information for operations issued by honest users on objects that no compromised user is authorized to access. Furthermore, $\mathcal{S}$ does not see the membership of the system (public keys of users) or even know how many users exist in the system, apart from corrupt/maliicous users.

E.1.3 Limitations of our Formalization

Although our cryptographic formalization is useful to prove Ghostor's anonymity, there are some aspects of Ghostor that it does not model. First, we do not directly model the anonymous payment (e.g., Zcash) aspect of Ghostor. Instead, we assume the existence of an ideal functionality for Zcash, $\mathcal{F}_{\text{Zcash}}$, that can be queried to validate payment (i.e., learn how much was paid and when). Second, we do not directly model network information (e.g., IP addresses) leaked to the server when clients connect, because this is hidden by the use of an anonymity network like Tor (§8). Third, whereas the Ghostor system allows operations to be processed concurrently (i.e., round trips of different operations may be interleaved), our formalization assumes that the Ghostor server processes each operation one at a time. Fourth, we do not fully model Ghostor's integrity mechanisms, such as the return value of `obtain_digests`.

Users may also be malicious (i.e., controlled by the adversary). In our formalization, the adversary may compromise users, but we restrict the adversary to doing so *statically*. This means that the adversary compromises users at the time of their creation. The environment $\mathcal{Z}$ may choose to give the adversary control over certain users and clients to try and distinguish the ideal world from the real world.

E.2 Ideal World

We define an ideal functionality for an anonymous object sharing system in the simulation paradigm, which captures Ghostor's privacy guarantee. Our notation and setup are as follows. The environment $\mathcal{Z}$ interacts with the party P representing a Ghostor client, which simply relay messages to the ideal functionality $\mathcal{F}$. The ideal-world adversary $\mathcal{S}$ interacts with $\mathcal{F}$.

E.2.1 Execution in the Ideal World

Control begins with the environment $\mathcal{Z}$. The environment may request P to initiate an operation provided by Ghostor's Client API: `GET`, `PUT`, `set_acl`, `create_user`, `obtain_token`, or `obtain_digests`. This is done via Initiate and New_User messages. In the ideal world, the P is a *dummy party*, which forwards these Initiate and New_User messages to $\mathcal{F}$.

We model `create_object` as a special case of `set_acl`. We find this convenient because both `create_object` and `set_acl` set the object's header. Furthermore, our implementation (§8) uses the same RPC call to handle both.

To perform certain operations (e.g., `GET`, `PUT`, `set_acl`, etc.), a user keypair is necessary. This user keypair can be used for asymmetric encryption/decryption with a key-private encryption scheme, and is used in order to obtain the object's signing key from the object header. To formalize this, we draw a distinction between *users* and *clients*. Users have keypairs and are represented in the ideal world with IDs; in contrast, the client is P. Each Initiate message contains the user_ID of the user on whose behalf the operation will be performed. That there is only client that will actually perform the operation informally captures the guarantee given by the anonymity network, that the server cannot tell apart different Ghostor clients on the basis of network information.

In summary, each Initiate message contains:

- user_ID specifying which user's keypair to use for this request
- opcode, which can be one of `GET`, `PUT`, `set_acl`, `create_user`, `obtain_token`, or `obtain_digests`
- new_contents if opcode = `PUT` or opcode = `set_acl`
- new_header if opcode = `set_acl`
- payment_ID (forwarded to $\mathcal{F}_{\text{Zcash}}$) if opcode = `obtain_token`
- object_ID specifying the object on which this request operates
- Payment token to fund the operation (if applicable)

No information related to proof of work is included because $\mathcal{S}$ will be able to simulate it without any external information. Upon receiving an Initiate message, $\mathcal{F}$ reveals some information to $\mathcal{S}$, described in Appendix E.2.2.

As mentioned earlier, we allow users to be corrupted, but require corruption to be static: users are corrupted at the time they are created. This is handled by the New_User message, which contains:

- inform, a bit indicating if the adversary is aware of this user
- compromise, a bit indicating if this user is corrupted or not

Upon receiving a New_User message, $\mathcal{F}$ generates a random user_ID, and keeps track of whether the user is compromised. If the inform bit is set, then the user_ID is given to the adversary $\mathcal{S}$ so that malicious users may add this user to ACLs. If the user is compromised, then $\mathcal{F}$ uses this information to give more information to $\mathcal{S}$ when processing requests (see Appendix E.2.2). In each `PUT` operation, $\mathcal{F}$ generates a fresh ID, denoted content_ID, to represent the contents being written to that object. We refer to this mapping from `PUT` operation to content_ID as the *content table*.

E.2.2 Information that $\mathcal{F}$ gives to $\mathcal{S}$

Each Initiate message that the dummy party P sends to $\mathcal{F}$ represents an API call (§2) to the server. Given each API call, $\mathcal{F}$ processes the request and reveals some information to $\mathcal{S}$. First, $\mathcal{F}$ checks if the user issuing the request is malicious or not. If the user is malicious, then $\mathcal{F}$ reveals to $\mathcal{S}$ all information about the request, including which user makes the request and all arguments to the request. If the user issuing the request is honest, then $\mathcal{F}$ reveals to $\mathcal{S}$ the opcode and the following information:

- For `create_user`, the user_ID is given to $\mathcal{S}$ if either the inform or compromise bits are set. Otherwise, nothing is given to $\mathcal{S}$.
- For `GET`, $\mathcal{F}$ gives $\mathcal{S}$ only the object_ID of the object being accessed. $\mathcal{S}$ gives back to $\mathcal{F}$ the content_ID of the content to be returned, or $\perp$ if the operation fails or is aborted by $\mathcal{S}$.
- For `PUT`, $\mathcal{F}$ gives $\mathcal{S}$ only the object_ID of the object being accessed, and the content_ID and *length* of the object contents being written. However, if a malicious user has ever been on the ACL of the object, the object contents are given to $\mathcal{S}$ in cleartext.
- For `set_acl`, $\mathcal{F}$ scans the ACL being set, identifying which users are malicious. For each honest user in the ACL, $\mathcal{F}$ replaces the corresponding rows of the ACL with NULL. As object is being re-encrypted, $\mathcal{F}$ either gives $\mathcal{S}$ a content_ID and length, or the cleartext contents, depending on whether a malicious user has ever been on the ACL of the object.
- For `obtain_token`, $\mathcal{F}$ reveals to $\mathcal{S}$ the payment_ID. $\mathcal{S}$ responds with tokens that can be redeemed with future operations. $\mathcal{F}$ returns integers back to the party, which can be used as payment tokens in future Initiate messages to pay for operations. $\mathcal{F}$ keeps track of which of these tokens are spent, based on feedback from $\mathcal{S}$ indicating for which operations the payment was accepted.
- For `obtain_digests`, $\mathcal{F}$ reveals to $\mathcal{S}$ the epoch number and object_ID for which digests are to be obtained.

Additionally, $\mathcal{F}$ checks that the payment token provided in the Initiate message is valid, and reveals to $\mathcal{S}$ a single bit indicating whether a valid token was provided.

We have not yet specified what $\mathcal{F}$ returns to P. In order to allow the adversary to make arbitrary integrity violations during an epoch, the return value must originate from $\mathcal{S}$. For `GET`, $\mathcal{S}$ returns the content_ID for the returned content; $\mathcal{F}$ translates it back into actual content and gives it to the party P who requested it. For `obtain_token`, $\mathcal{F}$ forwards the response from $\mathcal{S}$ back to P. For operations involving token payment, $\mathcal{S}$ gives $\mathcal{F}$ a bit indicating whether the payment was accepted, which is forwarded to the original party P. For operations performed by a malicious user, P gives $\mathcal{Z}$ the result of the operation.

At any time, $\mathcal{S}$ can send $\mathcal{F}_{\text{Zcash}}$ a payment_ID. If it does so, it will receive from $\mathcal{F}_{\text{Zcash}}$ a response message indicating if the payment to the server is valid, and if so, and how much was paid and when.

E.3 Real World

The real world models Ghostor's execution. We will prove that our model of Ghostor in the real world reveals essentially the same information to the adversary as is revealed to the adversary in the ideal world.

The real world has the following key differences from the ideal world, in order to properly model Ghostor's execution:

- The party P handles Initiate messages from $\mathcal{Z}$, instead of simply forwarding them to $\mathcal{F}$.
- The party P sends Request messages to $\mathcal{A}$ and receive Response messages from $\mathcal{A}$ (instead of $\mathcal{F}$).

- The party P encrypts object headers and object contents, and $\mathcal{A}$ receives the ciphertexts, according to the Ghostor protocol.

Upon receiving an $\mathsf{Initiate}$ message from $\mathcal{Z}$, the P performs the operation specified in the $\mathsf{Initiate}$ message by interacting with $\mathcal{A}$ according to the Ghostor protocol (Appendix A). We do not specify the protocol in additional detail here because it is already specified in Appendix A. Upon receiving a $\mathsf{New\_User}$ message, P creates a keypair $(\mathsf{pk}, \mathsf{sk})$ and generates a $\mathsf{user\_ID}$ for the new user and stores them locally. If the $\mathsf{compromise}$ bit is set, it shares the secret key with $\mathcal{A}$, and if either the inform or $\mathsf{compromise}$ bits are set, then it informs $\mathcal{A}$ of the $\mathsf{user\_ID}$ and public key. As in the ideal world, malicious users' results are given to $\mathcal{Z}$.

For `obtain_token`, recall that we model Zcash as an ideal functionality $\mathcal{F}_{\text{Zcash}}$, which allows the adversary to validate a payment transaction via Zcash and learn how much was paid and when. Although $\mathcal{A}$ *may* follow the protocol in Appendix A at times, it is not obligated to; it may violate the protocol in ways that are not immediately detectable to the clients. $\mathcal{Z}$ can also create users via $\mathsf{New\_User}$ messages, which are handled locally by P. They generate the corresponding keypair and locally store which $\mathsf{user\_ID}$ maps to that keypair. If the $\mathsf{New\_User}$ message has the inform bit set, then the $\mathsf{user\_ID}$ and pk for that user are given to $\mathcal{A}$; if the $\mathsf{compromise}$ bit is set, then $\mathcal{A}$ is also given sk for that user.

E.4 Simulator

We now describe a simulator $\mathcal{S}$ that, given any real-world adversary $\mathcal{A}$, performs the same attack in the ideal world as $\mathcal{A}$ does in the real world, by invoking $\mathcal{A}$ as a black box. Note that $\mathcal{S}$, by the design of $\mathcal{F}$, is not given any user identities, yet needs to interact with $\mathcal{A}$ as *some* user. The key idea is that $\mathcal{S}$ simply creates a single "dummy" user keypair, and performs all interaction with $\mathcal{A}$ on behalf of honest users as that one user. The design of Ghostor is such that the server cannot distinguish this from a separate keypair being consistently used for each honest user.

$\mathcal{S}$ works by simulating a real world in which $\mathcal{A}$ exists as a black box. Recall that the real world consists of the parties P, $\mathcal{Z}$, and $\mathcal{A}$; for clarity, we use Q to refer to P in this simulated real world, to distinguish it from P in the ideal world.

E.4.1 State Maintained by $\mathcal{S}$

$\mathcal{S}$ maintains a pool of tokens to use. Successful calls to `obtain_token` contribute to this token pool, $\mathcal{S}$ stores tokens in this pool. For operations that require payment, $\mathcal{F}$ does not tell $\mathcal{S}$ which particular tokens to use, so $\mathcal{S}$ chooses tokens randomly from the pool.

$\mathcal{S}$ also maintains a *ciphertext table*. In the messages received, certain *encryptable* pieces of data (e.g., $\mathsf{content\_IDs}$) correspond to encrypted data in the actual Ghostor. To account for this, the ciphertext table maps each encryptable datum received by $\mathcal{S}$ to a fake ciphertext.

- The fake ciphertext corresponding to object contents is an encryption of a "zero string" of the same length as the object contents. The key used to encrypt the zero string is the same as the key normally used to encrypt object contents.[2]
- The fake ciphertext corresponding to a NULL entry in the object header is an encryption of a "zero string" of the same length as the plaintext object header entry, using the dummy user keypair.

E.4.2 Overview

Now, we explain how $\mathcal{S}$ interacts with $\mathcal{A}$ upon receiving information from $\mathcal{F}$. When $\mathcal{F}$ asks $\mathcal{S}$ to start an operation, it interacts with $\mathcal{A}$ over multiple round trips according to the Ghostor protocol via the simulated party, making sure to *blind* the request messages appropriately by replacing ciphertexts with fake ciphertexts. All object header entries corresponding to non-corrupt users are blinded; entries are created for them in the ciphertext table. The decision of whether to blind the object contents depends on whether a corrupt user has permission to read the object. Note that $\mathcal{F}$ has already determined this by the time it has sent the message to $\mathcal{S}$, and has NULLed object header entries for non-corrupt users and replaced data for each object not shared with corrupt users with an ID from its contents table. Therefore, $\mathcal{S}$ simply needs to create fake ciphertexts for object data that correspond to IDs in $\mathcal{F}$'s content table and for NULLed object header entries. Any object contents or object header entries that are not blinded are encrypted exactly as in the normal Ghostor system; $\mathcal{S}$ then forwards the ciphertexts to $\mathcal{A}$.

E.4.3 Simulator Functionality

Now, we describe the simulator more precisely. For operations that require payment, $\mathcal{S}$ verifies that the message it received from $\mathcal{F}$ indicates that a valid token were paid. Then it chooses a token randomly from its store, unblinds it, and uses it when interacting with $\mathcal{A}$. If the operation is successful, it marks the token as "used" so it is not chosen for a later operation.

`create_user`. Suppose $\mathcal{S}$ receives a message from $\mathcal{F}$ with a `create_user` opcode. If the $\mathsf{compromise}$ bit is set, then $\mathcal{S}$ generates a keypair $(\mathsf{pk}, \mathsf{sk})$ for this user and stores the mapping from the provided $\mathsf{user\_ID}$ to this keypair. If the inform bit or $\mathsf{compromise}$ bit is set, then $\mathcal{A}$ is informed of this $\mathsf{user\_ID}$, as if Q received a $\mathsf{New\_User}$ message.

`set_acl`. Suppose $\mathcal{S}$ receives a message from $\mathcal{F}$ with a `set_acl` opcode. $\mathcal{S}$ has the party Q perform a `set_acl` operation.

- If this operation creates the object, then $\mathcal{S}$ generates the keypairs for the object, and creates the encrypted key list for the object. $\mathcal{S}$ constructs each entry of the key list correctly in plaintext, and then encrypts each one as follows. If the entry corresponds to a malicious user, then it encrypts the entry using that user's public key. If the entry corresponds to an honest user, then it creates a fake ciphertext (encryption of zero string of the same length) using the honest

[2] $\mathcal{S}$ has access to this key because it executed `set_acl` for this object in the past.

keypair shared by all honest users and adds the mapping in the ciphertext table. Then it completes the operation using the resulting encrypted keylist.

- If this operation operates on an existing object, then $\mathcal{S}$ performs the operation using PSK (with a check if the owner is malicious). If the message from $\mathcal{F}$ includes a content_ID and length, then $\mathcal{S}$ has the same operation include a fake ciphertext for the re-encrypted object contents; otherwise if $\mathcal{F}$ includes the contents, then $\mathcal{S}$ encrypts it to produce the new data ciphertext. In both cases, the key to encrypt the object data is updated with a fresh one.

`PUT`. Suppose $\mathcal{S}$ receives a message from $\mathcal{F}$ with a `PUT` opcode. There are three cases:

- Suppose the `PUT` was performed by an honest user, and no malicious users have ever been on the ACL. $\mathcal{S}$ receives the ID of the object and the length of the contents being written. In the simulation, $\mathcal{S}$ has Q perform a `PUT` operation, using WSK. $\mathcal{S}$ uses a fake ciphertext (encrypted string of zeros of the correct length) and adds a mapping from the provided content_ID to the fake ciphertext in the ciphertext table.
- Suppose the `PUT` was performed by an honest user, but malicious users have been on the ACL of the object. $\mathcal{S}$ receives the ID of the object and the object contents. Then $\mathcal{S}$ encrypts the object contents and uses the resulting ciphertext instead of using a fake ciphertext, and has Q interact with $\mathcal{A}$ to write the fake ciphertext to the specified object.
- Suppose the `PUT` was performed by a malicious user. Then $\mathcal{S}$ has Q perform the operation using the information in the Initiate message, without using any fake ciphertexts.

`GET`. Suppose that $\mathcal{S}$ receives a message from $\mathcal{F}$ with a `GET` opcode. There are two cases:

- Suppose the `GET` was performed by an honest user. In this case, $\mathcal{S}$ gets the object_ID of the object being accessed. Then $\mathcal{S}$ has Q perform the `GET` operation using RSK. The ciphertext returned by $\mathcal{A}$ is translated back to a content_ID based on the ciphertext table (or decrypted if it is not a fake ciphertext), and given back to $\mathcal{F}$.
- Suppose the `GET` was performed by a malicious user. In this case, $\mathcal{S}$ gets the entire Initiate message used to initiate this operation. Then $\mathcal{S}$ has Q perform the `GET` operation using the keypair for that malicious user. The ciphertext returned by $\mathcal{A}$ is translated back to a content_ID based on the ciphertext table (or decrypted if it is not a fake ciphertext), and given back to $\mathcal{F}$.

`obtain_token`. Suppose that $\mathcal{S}$ receives a message from $\mathcal{F}$ with an `obtain_token` opcode. The message contains the payment_ID, which is forwarded to $\mathcal{A}$. The tokens produced by $\mathcal{A}$ are then collected by $\mathcal{S}$. $\mathcal{S}$ keeps the tokens from $\mathcal{A}$ in its global pool of tokens. Then $\mathcal{S}$ forwards identifiers for the tokens back to $\mathcal{F}$ as the return value. If $\mathcal{A}$ attempts to send a message to $\mathcal{F}_{\text{Zcash}}$ (as part of `obtain_token` or at any other time), then $\mathcal{S}$ sends the message to $\mathcal{F}_{\text{Zcash}}$ in the ideal world, and gives the response to $\mathcal{A}$ in simulation.

`obtain_digests`. Suppose that $\mathcal{S}$ receives a message from $\mathcal{F}$ with an `obtain_digests` opcode. The message is forwarded to $\mathcal{A}$.

Notably, this model does not include the *payment* phase in which the client initiates a Zcash transaction to transfer funds. Instead, we model Zcash as a trusted party, which the adversary cannot control. This ensures that the server learns nothing during the payment phase in the actual protocol. Formally, we define an ideal Zcash functionality $\mathcal{F}_{\text{Zcash}}$, which the adversary can use to check if a Zcash transaction ID is valid. $\mathcal{F}_{\text{Zcash}}$ reveals only the time of the transaction and the amount paid. Modeling Zcash (i.e., providing a real-world setup that realizes $\mathcal{F}_{\text{Zcash}}$) is out of scope for this work.

E.5 Proof Sketch

We are now ready to define Ghostor's anonymity. We denote the security parameter as κ throughout this paper.

Theorem 1 (Privacy in Ghostor)**.** *Suppose that in Ghostor, the data encryption scheme is CCA2-secure, the ACL encryption scheme is CPA-secure, the ACL encryption scheme is key-private, payment tokens are blind, and $\mathcal{F}_{Zcash}$ is an ideal functionality for Zcash. For every non-uniform probabilistic polynomial-time real-world adversary $\mathcal{A}$, there exists a non-uniform probabilistic polynomial-time ideal-world adversary $\mathcal{S}$ such that for every non-uniform probabilistic polynomial-time environment $\mathcal{Z}$, $\mathcal{Z}$ cannot distinguish the real world with adversary $\mathcal{A}$ from the ideal world with adversary $\mathcal{S}$.*

Proof. We shall demonstrate that for every real-world adversary $\mathcal{A}$, there exists an ideal-world adversary (simulator) $\mathcal{S}$ such that there exists no environment $\mathcal{Z}$ probabilistic polynomial-time in κ that can distinguish between interacting with the real world and interacting with the ideal world. Specifically, for an arbitrary real-world adversary $\mathcal{A}$, we construct an ideal-world adversary $\mathcal{S}$ that uses $\mathcal{A}$ as a black box to perform the same attack in the ideal world as $\mathcal{A}$ performs in the real world. $\mathcal{S}$ simulates an environment that is computationally indistinguishable from the real world, meaning that $\mathcal{A}$ will behave the same way in simulation with at most a negligible difference in probability. We take $\mathcal{S}$ as the simulator described in Appendix E.4.

There are two things to prove:

1. From $\mathcal{A}$'s perspective, the simulated world provided by $\mathcal{S}$ is computationally indistinguishable from the real world.
2. From $\mathcal{Z}$'s perspective, the real world with adversary $\mathcal{A}$ is computationally indistinguishable from the ideal world with adversary $\mathcal{S}$.

To show that these statements are true, we consider a sequence of seven hybrid setups. Although the two statements above are in principle separate, we use the same sequence of hybrids to prove both of them. Note that $\mathcal{H}_0$ is equivalent to the real-world setup, and $\mathcal{H}_6$ is equivalent to the simulated setup. In a true hybrid argument, only one operation can be modified at a time; our hybrids in the proof sketch below should be interpreted as key stages.

Hybrid $\mathcal{H}_0$. This is exactly the real-world setup in Appendix E.3.

Hybrid $\mathcal{H}_1$. This is the same as $\mathcal{H}_0$, except that we replace $\mathcal{A}$ with $\mathcal{S}$. $\mathcal{S}$, in this hybrid, maintains a simulated party Q corresponding to P, and internal to $\mathcal{S}$, these simulated parties interact with $\mathcal{A}$. P interacts with $\mathcal{S}$; when $\mathcal{S}$ receives a message from P, it forwards it to $\mathcal{A}$ via Q, and when $\mathcal{A}$ sends a message to one of $\mathcal{S}$'s simulated parties Q, it forwards it to P. Similarly, when $\mathcal{A}$ sends a message to $\mathcal{F}_{\text{Zcash}}$, $\mathcal{S}$ forwards the message to $\mathcal{F}_{\text{Zcash}}$, obtains the response, and forwards it to $\mathcal{A}$, as if $\mathcal{A}$ communicated with $\mathcal{F}_{\text{Zcash}}$ directly.

$\mathcal{S}$ acts simply as a relay, shuttling data back and forth between P and Q and between $\mathcal{A}$ and $\mathcal{F}_{\text{Zcash}}$. In particular, the messages observed by $\mathcal{A}$ and $\mathcal{Z}$ are exactly the same as before. Therefore, neither $\mathcal{A}$ nor $\mathcal{Z}$ can distinguish $\mathcal{H}_0$ from $\mathcal{H}_1$.

Hybrid $\mathcal{H}_2$. This is the same as $\mathcal{H}_1$, except that we now introduce the ideal functionality $\mathcal{F}$. $\mathcal{F}$, in this hybrid, just relays messages back and forth between the real-world party P and the simulator $\mathcal{S}$.

Here, the newly introduced $\mathcal{F}$ acts as another intermediate relay. Again, the messages observed by $\mathcal{A}$ and $\mathcal{Z}$ are distributed exactly the same as before. Therefore, neither $\mathcal{A}$ nor $\mathcal{Z}$ can distinguish $\mathcal{H}_1$ from $\mathcal{H}_2$.

Hybrid $\mathcal{H}_3$. We change P to a dummy party as in the ideal world. Instead, $\mathcal{S}$ handles participating in the protocol as the honest clients, including PoW. The requests for operations are forwarded by the party P to $\mathcal{F}$.

Although $\mathcal{S}$ now uses its dummy user keypair to interact with the server, the encryption is *key-private*; the server cannot distinguish an ACL entry encrypted under a user's key from the same ACL entry encrypted with $\mathcal{S}$'s dummy user key. Therefore, neither $\mathcal{Z}$ nor $\mathcal{A}$ can distinguish $\mathcal{H}_2$ from $\mathcal{H}_3$.

Hybrid $\mathcal{H}_4$. This is the same as $\mathcal{H}_3$, except that $\mathcal{F}$ replaces ACL entries of honest users with NULL; $\mathcal{S}$ replaces NULL entries with encryptions of zero under the dummy key, for the ACLs of the real-world protocol.

The *semantic security* of the encryption scheme used for ACLs guarantees that, to the adversary, an encryption of zero is indistinguishable from the actual encrypted ACL entry. Therefore, neither $\mathcal{Z}$ nor $\mathcal{A}$ can distinguish $\mathcal{H}_3$ from $\mathcal{H}_4$.

Hybrid $\mathcal{H}_5$. This is the same as $\mathcal{H}_4$, except that $\mathcal{F}$ also replaces object contents with IDs in its content table, and $\mathcal{S}$ in turn replaces these IDs with fake ciphertexts in its ciphertext table. In particular, if all users in an object's ACL are honest, then $\mathcal{F}$ and $\mathcal{S}$, together, replace the contents of the object with an encryption of the zero message of the same length, using the same key normally used to encrypt the object contents.

The *semantic security* of the encryption scheme used to encrypt object contents guarantees that $\mathcal{A}$ cannot distinguish between the fake ciphertext and the actual ciphertext. Furthermore, because the plaintext is returned as the result of the operation, we need to be sure that $\mathcal{A}$ cannot create a new valid ciphertext with a different plaintext distribution. Fortunately, the fact that we use CCA2-secure *authenticated encryption* guarantees this; the adversary cannot create a new ciphertext based on the fake one. Therefore, neither $\mathcal{A}$ nor $\mathcal{Z}$ can distinguish $\mathcal{H}_4$ from $\mathcal{H}_5$.

Hybrid $\mathcal{H}_6$. This is the same as $\mathcal{H}_5$, except that $\mathcal{S}$ keeps track of a pool of tokens, $\mathcal{S}$ gives $\mathcal{F}$ identifiers for the tokens, and $\mathcal{F}$ gives $\mathcal{S}$ a bit indicating if a valid token was used instead of specifying which token was used.

The *blindness* property of the blind signature scheme means that, to the server, different payment tokens, after being unblinded, are indistinguishable from each other. To the environment $\mathcal{Z}$, the interface is exactly the same and tokens are expended exactly as before. Therefore, neither $\mathcal{A}$ nor $\mathcal{Z}$ can distinguish $\mathcal{H}_5$ from $\mathcal{H}_6$. $\square$

F Ghostor's Integrity Guarantee

In this appendix, we state the integrity guarantee provided by Ghostor.

F.1 Linearizability

Before we formalize Ghostor's VerLinear guarantee, we define linearizability as a consistency property. Linearizability is well-studied in the systems literature [34,42], and providing a comprehensive survey of this literature and a fully general definition is out of scope for this paper. Here, we aim to define linearizability in the context of Ghostor, to help frame our contributions.

Definition 1 (Linearizability). *Let F be a set of objects stored on a Ghostor server, and let U be a set of users who issue read and write operations on those objects. The server's execution of those operations is* linearizable *if there exists a linear ordering L of those operations on F, such that the following two conditions hold.*

1. *The result of each operation must be the same as if all operations were executed one after the other according to the linear ordering L.*
2. *For every two operations A and B where B was dispatched after A returned, it must hold that B comes after A in the linear ordering L.*

In Ghostor, **an object's digest chain implies a linear ordering** L of `GET` and `PUT` operations, as follows.

Linear ordering L implied by a digest chain. The linear ordering L to which the server commits is based on the digest chain as follows. First, we assign a sequence number to write operations according to the order of their PREPARE digests in the digest chain. Next, we bind each operation to a digest in the digest chain as follows:

- Each read is bound to the digest representing that read.
- A write with sequence number i is bound to the first COMMIT digest whose sequence number is at least i. **This is either the COMMIT digest for this write, or the COMMIT digest for a concurrent write that wins over this one based on the conflict resolution policy in §5.4.**

Assuming the digest chain is well-formed (all cases except Case 1 below), each write will be bound to a COMMIT digest that is after its PREPARE digest and before or at its COMMIT digest. Finally, we generate the linear ordering as follows:

- If two operations are bound to different digests, then they appear in L in the same order as the digests appear in the digest chain.
- If two writes are bound to the same digest, then they are ordered in L according to their sequence numbers.

For example, suppose the digest chain contains $(R_1, P_1, R_2, P_2, R_3, C_2, R_4, P_3, R_5, C_1, R_6, C_3, R_7, P_4, R_8, C_4, R_9)$, where R denotes a read digest, P denotes a PREPARE digest, and C denotes a COMMIT digest. The corresponding linear ordering of operations is $L = (R_1, R_2, R_3, W_1, W_2, R_4, R_5, R_6, W_3, R_7, R_8, W_4, R_9)$, where R denotes a read operation and W denotes a write operation.

F.2 Verifiable Linearizability

We begin by stating and proving Theorem 2 below, which specifies the achieved guarantees when some users perform the verification procedure for an epoch. Then, we present the VerLinear property of Ghostor as Corollary 1, a special case of Theorem 2. We use this approach because Theorem 2, despite being a more general statement, has fewer edge cases than Corollary 1, and we feel its proof is easier to understand in isolation. The statement of Corollary 1 maps directly to our informal definition of verifiable linearizability in §3; the key differences are only that Corollary 1 is explicit that security depends on collision resistance of Ghostor's hash function and existential unforgeability of Ghostor's signature scheme, introduces variables that are useful in the proof, and states the security guarantee as the contrapositive of Guarantee 1.

Theorem 2 (Epoch Verification Theorem)**.** *Suppose the hash function H used by Ghostor is a collision-resistant hash function with security parameter κ. Let $\mathcal{B}$ be a non-uniform adversary that is probabilistic polynomial-time in κ performing an active attack on the server. Let E be a list of consecutive epochs. For each epoch $e \in E$, let U_e be a set of users for whom the verification procedure for a particular object F detected no problems during epoch e, and let O_e be the set of operations performed by those users on F.* ***If*** *$U_e \neq \varnothing$ (i.e., U_e is nonempty) for all $e \in E$,* ***then*** *there exists, with probability at least $1 - \mu(\kappa)$, where μ denotes a negligible function, a linear ordering L of operations in $O = \bigcup_{e \in E} O_e$ and possibly some other operations, such that for the users in U and their operations O, the following two statements hold.*

1. *The result of each successful operation is the same as if all operations were executed one after the other according to L.*
2. *For every two operations A and B where B was dispatched after A returned, B comes after A in L.*

Proof. We will perform a reduction to show that if there exists an adversary $\mathcal{B}$ that can cause one of the two conditions to be violated, then there exists an adversary $\mathcal{A}$ that can violate the collision-resistance of H with non-negligible probability. For concreteness, suppose that $\mathcal{B}$ performs such an attack with non-negligible probability $\delta(\kappa)$ (so that the condition in the theorem holds with probability $1 - \delta(\kappa)$). We will explain how $\mathcal{A}$ can succeed in finding a hash collision with non-negligible probability.

By the nature of the attack, $\mathcal{B}$ is able to violate the property in the theorem statement, while remaining undetected by users in U. Observe that $\mathcal{B}$'s attack must fall into at one of four cases.

1. There exists at least one object such that $\mathcal{B}$ does not commit to a valid digest chain for an epoch, for some honest user.
2. There exists at least one object such that $\mathcal{B}$ commits to a different digest chain for different honest users.
3. There exists an operation on an object $f \in F$ whose result is different from the result that would be obtained by applying the operations one after the other in the linear ordering implied by f's digest chain.
4. There exist operations a and b on the same object, where a was issued after b completed, but a precedes b in the linear ordering implied by the digest chain.

In particular, if $\mathcal{B}$'s attack does not fall into one of these cases, then the locality property proved in §3 of [42] guarantees that $\mathcal{B}$'s behavior is consistent with the theorem statement (linearizability of operations in L). We will show that no matter which of the above four cases describes $\mathcal{B}$'s attack, $\mathcal{A}$ can find a hash collision.

Case 1. In this case, $\mathcal{B}$ returns an invalid linear ordering to a user when the user performs an `obtain_digests` operation. The ordering could be invalid because the digest is not signed properly, or the digests do not form a well-formed chain. This also includes the case where a user's operation is missing from the digest chain. Because we require that $U_e \neq \varnothing$ for all $e \in E$, this will be detected with probability 1. Therefore, we do not consider this case.[3] An important note is that if each L_e is valid, then L is valid.

Case 2. In this case, the adversary returns different histories to different users. Because the histories differ, they cannot be the same in all epochs; we consider an epoch e in which they differ. This allows us to confine our argument to a single epoch. In particular, there exist two `obtain_digests` operations on the same object during epoch e, for which $\mathcal{B}$ returns different histories in a way that is not detectable.[4] We define two subcases.

In the first subcase, the leaf of the Merkle tree, containing the hash of the final digest for the object in the epoch, is differ-

[3]For the purpose of this proof, it does not matter which party signs the digest, only that it was signed with the correct signing key (which is a per-object key rather than a per-user key). In the actual Ghostor system, only an authorized user can produce the signature due to the existential unforgeability of the signature scheme.

[4]If for all $e \in E$ where the histories differ, only a single call is made to `obtain_digests`, then the server cannot commit to multiple histories, and therefore cannot attack the protocol in this way; therefore, we do not consider this case.

ent for each call. However, given our consistency assumption for the blockchain, each user will see the same Merkle root. Furthermore, because the leaves of the Merkle tree are sorted and each intermediate node indicates the range of objects in each of its children, each node in the root-to-leaf path unambiguously specifies the hash of the next node in the path. Because the first element (root) is the same for the paths returned in each call to `obtain_digests`, but the last element is different, there must be a hash collision somewhere along the path. $\mathcal{A}$ finds this collision.

In the second subcase, both calls to `obtain_digests` see the same Merkle leaf and therefore the same hash of the final digest, but see different digest chains regardless. Observe that the last digest and first digest, for this epoch's digest chain, are fixed based on the checkpoint for this epoch and the checkpoint for the previous epoch, which the client can obtain from the server (to make the argument simpler, we consider the final digest of the previous epoch to also be the first digest of the current epoch). Furthermore, the user knows the hashes of these digests, from the checkpoints on the blockchain. Therefore, if first or last digests of the digest chains returned to both calls to `obtain_digests` differ, then $\mathcal{A}$ can use them to find a hash collision (since their hashes must match the Merkle leaves). If these digests match, then the intermediate digests must differ. To find a collision in this case, $\mathcal{A}$ simply walks backwards along the digest chains, until they differ. $\mathcal{A}$ can use the digests on each chain, at the point that they differ, to obtain a hash collision.

Case 3. Observe that the result of any committed write is "Success." Therefore, we can restrict this case to *reads that return the wrong value*.

Suppose that a read operation in O_e (for some $e \in E$) returned a value that is not consistent with the linear ordering for the object. In order for the operation to be considered successful, the $\mathsf{Hash}_{\mathsf{data}}$ value in the signed digest received by the client must match the hash of the returned object contents. Furthermore, the verification procedure guarantees that the $\mathsf{Hash}_{\mathsf{data}}$ value in each digest corresponding to a read matches the $\mathsf{Hash}_{\mathsf{data}}$ value in the latest write at that time—it does this by checking that $\mathsf{Hash}_{\mathsf{data}}$ never changes as the result of a read, and that it only changes in the COMMIT digests of winning writes. It follows that the incorrect value returned by the read operation, and the correct value that should have been returned (which was written by the latest write), have the same hash. $\mathcal{A}$ can present these two values as a hash collision.

Case 4. If an operation is missing from the digest chain entirely, this will be detected by the client that issued the operation. We now consider the case where the digests appear in the wrong order. Concretely, let op_1 and op_2 be two operations, where op_2 is issued after op_1 completed. If op_1 is a `PUT`, then d_1 is its COMMIT digest; otherwise, if op_1 is a `GET`, d_1 is the single digest for that `GET`. If op_2 is a `PUT`, then d_2 is its PREPARE digest; otherwise, if op_2 is a `GET`, d_2 is the single digest for that `GET`. Because op_2 is issued after op_1 completed, their digests should unambiguously appear in order in the digest chain: d_1 appears before d_2. Now, suppose d_1 appears sometime after d_2, so that the linear ordering is inconsistent with execution order. In this case, $\mathcal{A}$ waits until the users have run the verification procedure, and then rewinds $\mathcal{B}$'s state to a point after $\mathcal{B}$ has committed op_1, but before op_2 has been issued. The client places a fresh nonce in d_2 this time around, but otherwise execution is resumed as before. $\mathcal{A}$ waits until the user runs the verification procedure again, and it compares the digest chains produced by $\mathcal{B}$'s execution both times. Because all that changed is the client's nonce in d_2, and it is taken from the same uniform random distribution, $\mathcal{B}$'s probability of performing a successful attack is still non-negligible. So the probability that $\mathcal{B}$ performed a successful attack in both distributions is non-negligible ($\delta(\kappa)^2$). In this case, $\mathcal{A}$ walks the digest chains backward starting at d_1; the digest chains must differ at some point, because d_2 precedes d_1 in the first history, d_2 has a different random nonce in the second history, and the digest for d_1 is the same in both histories. This way, $\mathcal{A}$ can obtain a hash collision. □

Although the two conditions in Theorem 2 are the same as those in Definition 1, Theorem 2 does *not* guarantee linearizability of operations in O (operations performed by users in U). This is because the linear ordering L in Theorem 2 includes *additional* operations in the system beyond those in O, which could be digests that the server replayed or operations performed by users who did not run the verification procedure. This motivates us to state Corollary 1, which specifies under what conditions a set of users can be sure that their operations were processed in a linearizable way. Because our definition is now in line with linearizability (Definition 1), we can leverage the locality property of linearizability [42] to state the corollary in terms of a single object.

Corollary 1 (Verifiable Linearizability)**.** *Suppose the hash function H used by Ghostor is a collision-resistant hash function and the signature scheme is existentially unforgeable. For any adversary probabilistic polynomial-time in κ, any object F, and any list E of consecutive epochs: suppose that for each epoch $e \in E$, the set U_e of users who ran the verification procedure on F during epoch e (1) is nonempty (i.e., $U_e \neq \varnothing$) and (2) contains all users who wrote the object F during epoch e (and possibly other users too). With probability at least $1-\mu(\kappa)$, where μ denotes a negligible function,* ***if*** *no user detects a problem when running the verification procedure,* ***then*** *the server's execution of operations in $O = \bigcup_{e \in E} O_e$ is linearizable, where O_e is the set of operations performed by users in U_e during epoch e.*

Proof. By Theorem 2, we know that there exists a linear ordering L containing all operations in O plus some other authorized operations on F such that Properties #1 and #2 in the statement of Theorem 2 hold for operations in O, with respect to L. Because each U_e contains all users who wrote

f during epoch e, and the signature scheme is existentially unforgeable, we know that all operations in L that are not in O must be reads. Let ℓ denote the subset of L consisting only of operations in O. Now, observe that Properties #1 and #2 in the statement of Theorem 2 also hold for the operations in O *with respect to* ℓ. This is because (1) L is the same as ℓ with some additional read operations, so the result of each operation, when operations are executed one after the other, is the same for both orderings, and (2) the relative ordering of operations in O is the same in both L and ℓ. Because ℓ contains only the operations in O and it satisfies Properties #1 and #2, it fulfills Definition 1. Therefore, the execution of operations in O is linearizable. $\square$

Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating

Shaghayegh Mardani*, Mayank Singh†, Ravi Netravali*
*UCLA, †IIT Delhi

Abstract

Despite the rapid increase in mobile web traffic, page loads still fall short of user performance expectations. State-of-the-art web accelerators optimize computation or network fetches that occur *after* a page's HTML has been fetched. However, clients still suffer multiple round trips and server processing delays to fetch that HTML; during that time, a browser cannot display any visual content, frustrating users. This problem persists in warm cache settings since HTML is most often marked as uncacheable because it usually embeds a mixture of static and dynamic content.

Inspired by mobile apps, where static content (e.g., layout templates) is cached and immediately rendered while dynamic content (e.g., news headlines) is fetched, we built Fawkes. Fawkes leverages our measurement study finding that 75% of HTML content remains unchanged across page loads spread 1 week apart. With Fawkes, web servers extract static, cacheable HTML templates for their pages offline, and online they generate dynamic patches which express the updates required to transform those templates into the latest page versions. Fawkes works on unmodified browsers, using a JavaScript library inside each template to asynchronously apply updates while ensuring that JavaScript code only sees the state that it would have in a default page load despite downstream content having already been loaded. Across a wide range of pages, phones, and live wireless networks, Fawkes improves interactivity metrics such as Speed Index and Time-to-first-paint by 46% and 64% at the median in warm cache settings; results are 24% and 62% in cold cache settings. Further, Fawkes outperforms recent server push and proxy systems on these metrics by 10%-24% and 69%-73%.

1 Introduction

Mobile web browsing has rapidly grown in popularity, generating more traffic than its desktop counterpart [18, 20, 57]. Given the importance of mobile web speeds for both user satisfaction [11, 12, 23] and content provider revenue [21], many systems have been developed by both industry and academia to accelerate page loads. Prior approaches have focused on pushing content to clients ahead of time [61, 70, 76, 19], compressing data between clients and servers [4, 67, 63], intelligent dependency-aware request scheduling [14, 42], offloading tasks to proxy servers [47, 65, 10, 6], and rewriting pages for the mobile setting (either by automatically serving post-processed objects to clients [71, 43, 51], or by manually modifying pages to follow mobile-focused guidelines [24, 34]). Yet despite these efforts, mobile page loads continue to fall short of user expectations in practice. Even on a state-of-the-art phone and LTE cellular network, the median page still takes over 10 seconds to load [7, 61].

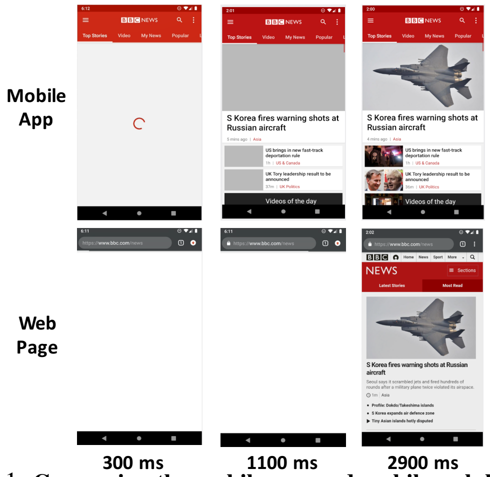


Figure 1: **Comparing the mobile app and mobile web browser loading processes for BBC News over an LTE cellular network.**

Our key observation is that, while existing optimizations are effective at reducing network fetch delays and client-side computation costs during page loads, *they all ignore a large and fundamental bottleneck in the page load process: the download of a page's top-level HTML file*. To fetch a page's top-level HTML, a browser often incurs multiple network round trips for connection setup (e.g., DNS lookups, TCP and TLS handshakes), server processing delays to generate and serve content, and transmission time. These tasks can sum to delays of hundreds of milliseconds, particularly on high-latency mobile links.[1] Only after receiving and parsing a page's HTML object can the browser discover subsequent objects to fetch and evaluate, make use of previously cached objects, or render any content to the blank screen. Thus, from a client's perspective, the entire page load process is blocked on downloading the page's top-level HTML object. This is true even in warm cache scenarios, since HTML objects are most often marked as uncacheable [44] (§2).

Eliminating these early-stage inefficiencies would be fruitful for two reasons. First, overall load times would reduce since client-side computation and rendering tasks for cached content could begin earlier and be overlapped with network and server-side processing delays for new content; the CPU is essentially idle as top-level HTML files are fetched in traditional page loads. Second, and more importantly, browsers could immediately display static content, rather than showing only a blank screen as the HTML is fetched (Figure 1). This is critical as numerous web user studies and recent performance metrics highlight user emphasis on content becoming visible quickly and progressively [46, 25, 35, 49, 66].

[1] These delays persist even for HTML objects served from CDNs since last-mile mobile link latencies still must be incurred.

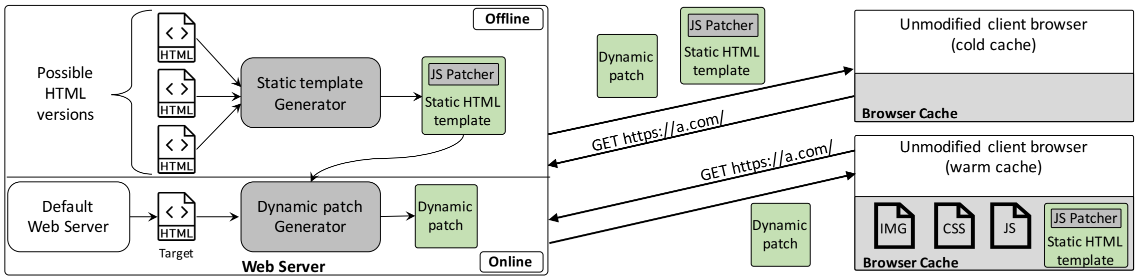


Figure 2: **Overview of cold and warm cache page loads with Fawkes. Servers return static, cacheable HTML templates, as well as uncacheable dynamic patch files that list the updates required to convert those templates into the latest page. Updates are performed dynamically using the Fawkes JavaScript patcher library that is embedded in the templates.**

To enable these benefits, we draw inspiration from mobile apps which, despite sharing many components with the mobile web (e.g., client devices, networks, content), are able to deliver lower startup delays (Figure 1). Apps reduce startup times by aggressively separating static content from dynamic content. At the start of executing a task (akin to loading a page), an app will issue a request for dynamic content *in parallel* with rendering locally cached content like structural templates, images, and banners. Once downloaded, dynamic content is used to patch the already-displayed static content.

Like apps, web pages already cache significant amounts of content across loads: 63% and 93% of the objects and bytes are cacheable on the median page. Yet startup times in warm cache page loads remain high due to download delays of top-level HTML files (§2). But why are HTML objects marked as uncacheable? The reason is that they typically bundle static content defining basic page structure with dynamic content (e.g., news headlines or search results). HTML which embeds dynamic content must be uncacheable so clients see the most recent page. Thus, at first glance, it appears that addressing this challenge with app-like templating would require a rethink of how web pages are written. However, our measurement study (§2) reveals that web pages are already highly amenable to such an approach given the large structural and content similarities for HTML objects across loads of a page. For instance, 75% of HTML tags on the median top 500 page have fixed attributes and positions across 1 week, and could thus be separated into static templates.

We present **Fawkes**, a new web acceleration system that modifies the early steps in the page load process to mirror that of mobile apps (Figure 2). Fawkes optimizes page loads in a two-step process. In the first phase, which is performed offline, web servers *automatically* produce static, cacheable HTML templates, which capture all content that remains unchanged across versions of a page's top-level HTML. The second phase occurs during a client page load; servers generate dynamic patches, which express the updates (i.e., DOM transformations) required to convert template page state into the latest version of a page. During cold cache page loads, browsers download precomputed templates while dynamic patches are being produced, and can quickly begin rendering template content and fetching referenced external objects as patches are pushed. In warm cache settings, browsers can immediately render/evaluate templates and referenced cached objects while asynchronously downloading the dynamic patch needed to generate the final page.

Realizing this approach with legacy pages and unmodified browsers requires Fawkes to solve multiple challenges:

- On the server-side, generating templates is difficult: traditional tree comparison algorithms [75, 36, 16, 54, 55] do not consider invariants involving a page's JavaScript and DOM state, but templates execute to completion prior to patches being applied and thus must be *internally consistent*. For example, removing an attribute on an HTML tag can trigger runtime errors if downstream JavaScript code accesses that attribute; an acceptable template must keep or omit both of these components. In addition, graph algorithms are far too slow to be used for online patch generation. Instead, Fawkes uses an empirically-motivated heuristic which trades off patch generation time for patch optimality (i.e., number of operations; note that the final page is unchanged). Our insight is that tags largely remain on the same depth level of the tree as HTML files evolve over time. This enables Fawkes to use a breadth-first-search variant which generates patches 2 orders of magnitude faster (in 20 ms) with comparable content.
- On the client-side, each static template embeds a special JavaScript library which Fawkes uses to asynchronously download dynamic patches and apply the listed updates. The primary challenge is in ensuring *view invariance* for JavaScript code that is inserted via an update: that code must see the same JavaScript heap and DOM state as it would have seen during a normal page load. For example, consider an update which adds a `<script>` tag to the top of the HTML template. If that script executes a DOM method that reads DOM state, the return value may include DOM nodes pertaining to downstream tags in the template—this state is already loaded in a Fawkes page load, but not in a default one, and may trigger execution errors. To provide view invariance, Fawkes uses novel shims around DOM methods which prune the DOM state returned by native methods based on knowledge of page structure and the position of the script calling the method.

We evaluated Fawkes using more than 500 real pages, live wireless networks (cellular and WiFi), and two smartphone models. Our experiments reveal that Fawkes significantly accelerates warm cache mobile page loads compared to default browsers: median benefits are 64% for Time-to-first-paint (TTFP), 46% for Speed Index (SI), 26% for Time-to-interactive (TTI), and 22% for page load time (PLT). Despite targeting warm cache settings, Fawkes speeds up cold cache loads by 62%, 24%, 20%, and 17% on the same metrics. Fawkes also outperforms Vroom [61] and WatchTower [47], two recent mobile web accelerators, by 69%-73% and 10%-24% on warm cache TTFP and SI. Importantly, Fawkes is complementary to these approaches; Fawkes with Vroom achieves Fawkes's TTFP and SI benefits, while exceeding Vroom's PLT improvements by 22%. Source code and experimental data for Fawkes are available at https://github.com/fawkes-nsdi20.

2 Motivation

We begin with a range of measurements that illustrate the startup discrepancies between mobile apps and web pages (§2.1), and the amenability of web pages to app-like templating (§2.2). Results used the LTE setup described in §5.1.

2.1 Mobile Apps vs. Mobile Web

We compare the load process of mobile apps and web pages by analyzing equivalent tasks across 10 web services; services were selected by randomly choosing web pages from the Alexa US top 100 list [5], and discarding those without a corresponding mobile app. Our corpus includes news, recommendation platforms, search engines, and social media applications. For each service, we equate loading a homepage with a mobile browser to loading the home screen with the mobile app. When applicable, we also compare equivalent searches on both platforms. We load each task in the mobile app and website back to back, and for each task, we log the time until the first paint to the screen and collect screenshots for three events: the first time either platform displays content to the user (Time-to-first paint, or TTFP), an intermediate checkpoint with additional displayed content, and the time when both platforms reach their final visual state. Mobile app screenshots and paint events are captured via the Apowersoft Recorder [8] and Android Systrace [31] tools, respectively. Since apps have content cached during installation, for fair comparison, we consider warm cache page loads (back to back). Certain mobile apps operate by displaying a logo for several seconds during startup, prior to displaying the home screen. We do not consider such apps here.

Startup delays are far lower with apps than web pages. Across the corpus, our experiments reveal that TTFP values are between 3.5×–5.2× lower with mobile apps than mobile web pages. Figure 1 provides a representative example of loading the home page for BBC News. As shown, despite the high network latencies and potential server processing delays, the mobile app is able to quickly display static content that establishes the overall app layout and logos in under 300 ms. We verified that the reason for this is that the app quickly pulls this content from its local cache while asynchronously fetching dynamic news headlines. In contrast, the BBC web page remains blank as the browser establishes a connection to the backend and downloads the top-level HTML for the page. Only upon receiving the HTML object can the browser begin rendering any static or dynamic content to the screen–this does not begin until 1200 ms, 4× longer than the app.

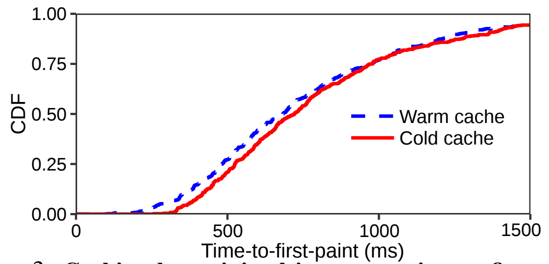


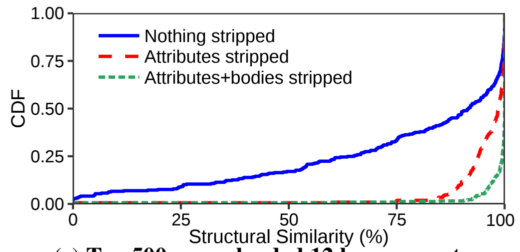
Figure 3: **Caching has minimal impact on time-to-first-paint since browsers cannot render cached content until they download the top-level HTML (typically uncacheable).**

Problem: uncacheable HTML limits caching benefits for the web. The above discrepancies between mobile apps and web pages are indicative of a fundamental difference in the startup tasks on the two platforms. Web HTML objects are used to set the context for the remainder of a page load, establishing render and JavaScript engine processes, creating a DOM tree (i.e., the browser's programmatic representation of the page's HTML) and JavaScript heap, and so on. However, most HTML objects are marked as uncacheable. For example, 72% are uncacheable across back to back loads of the top 500 pages; this number jumps to 85% for loads separated by 5 minutes. As a result, browsers are unable to make use of other objects marked as cacheable (e.g., images, CSS) until they download an HTML object; for reference, 53% and 93% of objects and bytes are cacheable on the median page. Figure 3 illustrates this point: median TTFP values are only 5.3% lower in warm cache scenarios than during cold cache page loads despite so many objects being cached.

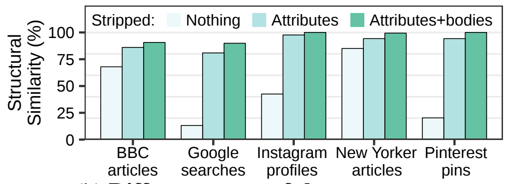
2.2 Templating Opportunities for the Web

Motivated by the startup discrepancies between mobile app and web page loads described above, we investigated how amenable web pages are to app optimizations. Our analysis focuses on the feasibility of extracting static templates from HTML objects that can be cached across page loads. We consider two different sets of sites: the Alexa top 500 landing pages [5], and a smaller set which includes 10 pairs of different pages of the same type (e.g., different news articles or search results). More dataset details are provided in §5.

We loaded each page (or pair of pages in the smaller corpus) multiple times to mimic different warm cache scenarios: back to back, 12 hours apart, 24 hours apart, and 1 week apart. In each setting, we compared the resulting top-



(a) **Top 500 pages loaded 12 hours apart.**



(b) **Different pages of the same type.**

Figure 4: **Structural similarity for HTML files over time. Similarity is defined as the percentage of shared tag sequences (including tag attributes, bodies, and types).**

level HTML objects to determine structural similarities. We identify each tag as a tuple consisting of its tag type (e.g., `<div>`), HTML attributes (e.g., `class`), and body (e.g., inline script code). Since static templates can be patched during page loads, we also consider tuple versions with all tag attributes stripped, and with both tag attributes and bodies stripped. Additionally, since HTML can be modeled as a tree where ordering matters, for each tag T, we generate a sequence of tags by following parent tags up from T to the root node. We then define structural similarity as the fraction of sequences that remain identical across the HTML versions.

Opportunity: HTML structure and content is largely unchanged over time. Figure 4 shows that HTML objects exhibit high structural similarity. For example, for the median top 500 page, 92% of HTML tags remain identical across loads separated by 12 hours; these numbers jump to 98% and 100% when attributes are stripped alone or with bodies. These trends persist for different pages of the same type. For instance, two different Instagram profile pages exhibit structural similarity of 98% when only attributes are stripped. The trends also persist for other time windows. For example, median similarities in the 1-week setting are 75% and 95% with nothing and attributes stripped, respectively.

Key Takeaways:

- Mobile apps exhibit a desirable startup process compared to mobile web pages because apps explicitly separate static and dynamic content, and immediately render cached static content while dynamic content is fetched. Web pages, on the other hand, remain blocked (blank screen) on downloading uncacheable HTML objects, despite most other objects being cacheable.
- Mobile web pages are amenable to app-like templating of static content since HTML objects (typically uncacheable) have large structural similarities over long time periods.

3 Design

Figure 2 shows the high-level design of Fawkes. Clients use unmodified browsers to load pages as normal. On the server side, websites must run Fawkes to handle incoming client HTTP(S) requests. The server-side Fawkes code performs two primary tasks. For a given page, Fawkes statically analyzes possible variants of the unmodified top-level HTML objects for the page and extracts a single **static HTML template** which maximally captures shared HTML content across versions. The generation of the static HTML template is performed offline, i.e., not during a client page load. Then, when a user loads the page, Fawkes compares the static HTML template to the target HTML, or the one that the default web server would have served without Fawkes, and generates a **dynamic patch**, which is a JSON file with an ordered list of DOM updates required to convert the template page into the target one.

The static HTML template includes an inline **JavaScript "patcher" library** that asynchronously downloads the dynamic patch file, and upon receiving it, dynamically applies the listed updates. During cold cache loads, Fawkes's server first returns the cacheable, static HTML template, and then streams the dynamic patch with HTTP/2 server push soon after; the template is sent earlier since it is precomputed, and this allows the browser to quickly start rendering template content and fetching referenced external resources. During warm cache page loads, the browser immediately begins to evaluate and render the cached template and other cached objects that it references as the patch downloads.

3.1 Server-Side Operation

In order to generate static HTML templates, Fawkes's server-side component leverages state-of-the-art tree matching algorithms [54, 55]. The goal of these algorithms is to determine the minimum distance between two (or more) tree structures; recall that HTML files are structured as trees (§2). In particular, these algorithms take as input a set of trees whose nodes are assigned labels. The algorithms then compute a set of operations that, if applied, would efficiently transform the first input tree into the second. Operations typically comprise three primary types: *delete* operations remove a node and connect its children to its parent, *insert* operations add a node to a specific position in the tree, and *rename* operations do not change node positions but instead only alter a node's label. Algorithm execution works much like string edit distance techniques, using dynamic programming and assigning each operation a cost of 1.

Altering tree matching algorithms: Fawkes must alter existing tree matching algorithms in several ways to ensure that they are compatible with HTML and web semantics. First, existing algorithms require each tree node to be labeled with an individual string. However, HTML tags can include state beyond simple tag type (e.g., `<div>` or `<link>`), each of which could be shared across versions of a page. Properties include attributes (e.g., `class`) that control the tag's behavior with respect to CSS styling rules and interactions with JavaScript code, and bodies such as inline JavaScript code or text to print. Failing to consider attributes during HTML comparison can result in either broken pages if attributes are incorrectly treated as equivalent, or suboptimal templates if shared attributes are not maximally preserved, i.e., any attribute discrepancy would require omitting a tag. Thus, Fawkes's tree comparison algorithm labels each HTML tag with a (type, [attributes], body) three-tuple.

Second, Fawkes opts to not support *rename* operations, and instead only supports new *merge* operations. Unlike rename operations that can entirely change a node's label to deem it equivalent to a node in the other tree, merge operations can only alter a tag's attributes or body to claim such equivalence. Importantly, merge operations do not allow tag types to be modified. The reasoning behind this decision is that different tags impose different semantic restrictions on HTML structure. For instance, an `<img>` tag is self-closing, and cannot contain children tags, while `<div>` tags can have arbitrary children structures. Rewriting a `<div>` tag to an `<img>` tag would thus trigger cascading effects on existing children tags, leading to smaller templates.

Generating static HTML templates: Fawkes uses the above tree matching framework to generate static HTML templates from a set of HTML files. We describe how to get this input set in §3.3, and for simplicity, describe the approach assuming two input HTML files. To start, Fawkes runs its tree matching algorithm to generate a set of operations which, if applied, would convert *HTML1* to *HTML2*. Fawkes then iterates through HTML1 and selectively applies certain updates to only keep content that is shared across the inputs. Delete operations are directly applied to HTML1 as they represent content which is not shared across versions and thus should not be part of the static HTML template. Similarly, insert operations are ignored as they represent content that must be added to reach HTML2 and is thus not shared. Finally, Fawkes strips all content (tag attributes and bodies) referenced by merge operations as these highlight discrepancies between HTML versions.

While applying these operations and generating static HTML templates, Fawkes must be careful to preserve page semantics and not violate inherent dependencies between page state. In particular, Fawkes must ensure that static HTML templates are *internally consistent* and do not trigger JavaScript execution errors when parsed; this is important as templates are parsed to completion prior to any patch updates being applied. The key challenge is that altering an HTML tag's attributes or body can have downstream effects due to the shared state between JavaScript code and the DOM tree [42]. For example, a downstream `<script>` tag may access an upstream `<p>` tag's attribute. Deleting that `<p>` tag's attribute can thus trigger execution errors when the browser reaches the `<script>` tag. Similarly, different `<script>` tags can share state on the JavaScript heap. As a simple example, an upstream tag may define a variable which the downstream tag accesses. Thus, cutting the upstream tag's body can trigger downstream execution errors.

Existing tree matching algorithms are unaware of such dependencies and are agnostic to the HTML execution environment. Thus, Fawkes applies a post-processing step to ensure that such dependencies are not violated in the static HTML template. Fawkes essentially iterates through the static HTML template, and upon detecting an altered tag, cuts downstream `<script>` tags. Fawkes could leverage techniques like Scout [42] to more precisely characterize the dependencies between tags and JavaScript code in an effort to preserve more state in static HTML templates. However, accurately capturing such fine-grained dependencies would require web servers to also execute HTML content and load pages. Our empirical results motivate that templates derived from static tree analysis sufficiently keep the browser occupied with render and fetch tasks as dynamic patches are fetched, obviating the need for dynamic processing.

Generating dynamic patches: Fawkes servers must generate dynamic patches that list updates which, if applied, would convert the page state produced by a template into its desired final form. The inputs for patch generation are the static HTML template and the target HTML which is the file that a default web server would serve during the current page load.

The tree comparison algorithm described above can produce the desired set of transformation updates that a patch must contain. However, such algorithms are far too slow for patch generation which, unlike template generation, must be performed online, during client page loads. Thus, Fawkes uses a tree comparison heuristic which trades off patch generation time for optimality in terms of number of operations in the patch. The key insight is empirically motivated: we observe that tags most often remain on the same depth level of a tree as HTML files evolve over time. Our analysis of HTML files for 600 pages over a week revealed that, at the median and 95th percentile, 0% and 1% of tags in target HTML files were at a different depth than they were a week earlier. This property favors a breadth-first-search approach over a depth-first-search one, and implies that we need not consider new positions for a tag outside of its current level (as traditional algorithms would). So, for each level in the target HTML file, Fawkes's algorithm works as follows:

1. Create hash maps for both the target and template HTML files that list all of the nodes for a given tag type in the order they appear on that level (from left to right).
2. Iterate through the template's level from left to right and handle each node in one of two ways. If the node's tag type exists in the same level of the target, match this node to the closest node of the same type with the same parent, and remove that node's entry from the target's hash map; if no node has a matching parent, match to the closest node of the same type. Record a *merge* op-

eration by comparing the attributes and bodies for the matched nodes. Else, if the tag type does not exist in the same level of the target, delete this node in the template and record a *delete* operation.

3. Once we reach the end of the template's level, apply *move* operations to order all matched nodes in the template in the same way as they appear in the target; objects which remain in the same position do not require any operation. Note that *move* operations (which simply change the position of a node) are not supported by traditional tree diffing algorithms, and are only enabled by our heuristic's "look ahead" hash maps. Also, note that moves made at this level are immediately reflected in lower levels of the tree as children are reordered.
4. Finally, from left to right, insert any remaining nodes listed in the target's hash map to the appropriate position. Record *insert* operations for these additions.

The key limitation of this heuristic is with respect to nodes moving across levels in the tree. Traditional algorithms can identify such cases, while Fawkes's approach would automatically require a delete at the original level and an insert at the new level. However, as noted above, such transformations are rare. In addition, matching nodes to their closest counterparts with the same parent could be suboptimal: an inserted node in the target can create cascading suboptimal rename and move operations for nodes of that type. Despite such potential inefficiencies, correctness of the final page load is unaffected. We compare this heuristic to standard tree comparison algorithms and other heuristics in §5.5.

Each update in a dynamic patch must identify a node to which the update should be applied. Fawkes identifies DOM nodes by their child paths from the root of the HTML tree. For example, a child path of [1,3,2] represents an HTML tag that can be reached by traversing the first child of the root HTML tag, the third child of that tag, and then the second child of that tag. Child path ids are easy to compare and can be computed purely based on HTML tree structure.

3.2 Client-Side Operation

To load a page, a mobile browser first loads the static HTML template, whose initial tag is the Fawkes patcher JavaScript library. The patcher begins by issuing an asynchronous XMLHttpRequest (XHR) request for the page's latest dynamic patch. The patcher defines a callback function on the XHR request which will be executed upon receiving the dynamic patch JSON file to apply updates. The patcher then defines the DOM shims required by the callback to apply the dynamic patch updates (described below). Finally, the patcher removes its HTML `<script>` tag from the page to prevent violating downstream state dependencies and to ensure that the final page's DOM tree is unmodified. We note that the state defined by the patcher persists on the JavaScript heap despite its tag being removed from the page.

Applying updates: Upon downloading the dynamic patch, the patcher's callback function iterates over the listed updates and applies them in order until completion. To apply a given update, the patcher first obtains a reference to the affected DOM node (i.e., the one listed in the update) by walking the DOM tree based on the listed child path. The patcher then uses native DOM methods to apply the update.

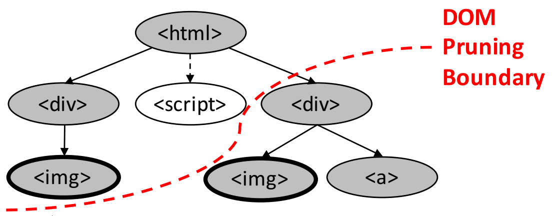


Figure 5: **Update challenge 1: provide view invariance to JavaScript code by hiding downstream DOM state. Shaded nodes are part of the static HTML template. The `<script>` tag is inserted via Fawkes's patcher and calls a DOM method to find `<img>` tags. The native DOM method would return the two nodes outlined in bold, even though the rightmost one would not be returned in the default page load; Fawkes's DOM shims prune the rightmost node from the return value.**

For insert operations, the patcher first creates a new DOM node using document.createElement(), sets the appropriate attributes with Element.setAttribute(), and then adds it to the appropriate position in the DOM tree by calling document.insertBefore(). Adding nodes to the DOM tree can have cascading effects with respect to rendering and layout tasks (both of which are expensive). To mitigate these overheads, Fawkes intelligently looks ahead in the update list to determine if subsequent updates reference the node being added by the current insert update [43, 22]. In these cases, Fawkes constructs a DOM subtree on the JavaScript heap prior to applying the entire subtree to the actual DOM. Fawkes uses similar techniques to handle merge and delete operations.

Handling DOM discrepancies: There are two main challenges with applying updates, both of which relate to JavaScript execution and its interaction with the DOM tree. Fawkes handles both using a novel set of shims (or wrappers) around DOM methods, which are the vehicles with which JavaScript can access or modify the DOM tree.

- The first issue is with providing *view invariance* for JavaScript code inserted via an update: that code must see the same JavaScript heap and DOM state as it would have seen during a normal page load. This is challenging since updates are not applied until after a page's static HTML template is entirely parsed. For example, consider Figure 5 where an inserted `<script>` tag invokes a DOM method to read `<img>` tags in the page. The return value for this method would include an `<img>` tag that is downstream in the page's HTML; this divergence from the default page load could trigger JavaScript execution errors or alter page semantics. To handle this, Fawkes shims all DOM methods which return a DOM node or a list of DOM nodes; examples include document.getElementById() and document.getElementsbyTagName(). Each shim calls the native method and prunes the result prior to returning it to the client. Pruning is done by identifying the position in the DOM tree of the script invoking the DOM method, and then removing DOM nodes in the result which are below that position in the DOM tree. Fawkes's shims skip pruning for callback functions (e.g., timers) and provide *view plausibility* since the page makes no guarantee on what DOM state those asynchronous events can encounter. We note that it is not possible for inserted scripts to see less DOM state than it would in a default page load because updates are ordered with respect to HTML positions.

- The second issue is that JavaScript code can alter the DOM tree in ways that affect the child path ids for subsequent updates. The reason is that the child path ids listed in the dynamic patch are based on the static HTML, which does not consider JavaScript execution, but are applied to the dynamic DOM tree which JavaScript code can manipulate (Figure 6). To handle this, Fawkes shims DOM methods that affect DOM structure, either by adding, removing, or relocating nodes; example methods include document.appendChild() and document.insertBefore(). Each shim calls the native method, logs its effect on DOM structure (e.g., the child path of an added node), and then returns the value. When the patcher attempts to apply an update, it first checks this log, and modifies the child paths in the remaining updates based on the listed DOM changes. We note that JavaScript can also invalidate a listed update, e,g,. by replacing a `<div>` tag with an `<img>` tag in the same position. Fawkes's shims detect these alterations, and the patcher discards such updates since JavaScript takes priority over HTML for final page structure.

3.3 Identifying HTML Objects to Consider

Fawkes's server-side static template generation inherently relies on having a set of representative HTML files from which to extract a template. Here we discuss several approaches for websites to generate this input set for each of their pages; Fawkes is agnostic to the specific approach that a site uses for this. We note that the input set need not be comprehensive and cover all possible HTML versions for a page since patches will include all necessary updates to reach the target page. However, considering a comprehensive set of HTML objects can reduce the number of updates required at runtime, leading to improved performance.

Option 1: empirical analysis: One approach is for web servers to log the HTML objects that they would serve to clients over time without Fawkes. Fawkes can then periodically recompute a static HTML template based on the latest served HTML files to account for structural modifications that developers make to the page. An advantage of this approach is that the static HTML templates will inherently be based off of the popular pages that are actually served to clients. For instance, if a very rare state configuration would

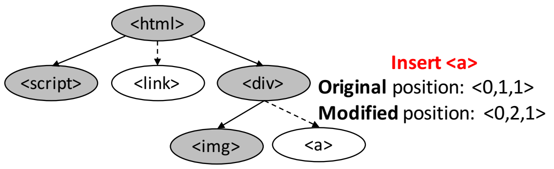


Figure 6: **Update challenge 2: revise update positional information to reflect JavaScript execution. Shaded nodes are part of the static HTML template. Upon execution, the `<script>` tag inserts an adjacent `<link>` tag into the page. Later, when the patcher tries to apply an update to insert an `<a>`, the listed child path id has gone out of date and must be updated.**

alter the structure of a page, most page loads would benefit from *not* considering this version in template generation.

Option 2: leveraging web frameworks: An alternative approach is to leverage the model-view-controller architecture that many popular web frameworks (e.g., Django, Ruby on Rails, and Express) use. In these systems, incoming requests are mapped to a controller function which generates a response by executing application logic code that combines application data and premade HTML blocks. Note that these blocks are small, spanning only a few tags each, and are significantly augmented with HTML tags generated by application logic–this precludes us from using these blocks as our templates. To leverage this structure, we can perform standard static program analysis [62, 37] on application code (particularly the controller for the URL under consideration) to determine the possible HTML block combinations and dynamically produced HTML code that could result for a page.

Option 3: hybrid approach: A final approach is to perform static program analysis on the application backend source code to determine what inputs affect HTML structure, e.g., Cookie values, database state, time of day, etc. Fawkes can then simply probe the backend with different input values to generate a range of potential HTML objects that could be returned to clients; static HTML template generation would then work in the same way as Option 1.

Case studies: Our evaluation (§5) primarily focuses on Option 1. However, to validate the feasibility of the remaining options, we analyzed the source code of two real open source web applications: Reddit [2] and ShareLatex [1]. Both applications follow the MVC model described above, with Reddit using the Python Pylons framework [58] and ShareLatex using NodeJS's Express [50]. For both applications, we wrote custom static analyzers which profile the controller for the sites' landing pages. The output of the profiler is an intermediate template that intertwines HTML code with Python (or JavaScript) logic that, when executed, reads in application variables and outputs a fully formed HTML file. Following branch conditions and unrolling loop bodies in the intermediate template revealed that a ShareLatex project page has 16 possible HTML structures, while Reddit can have over 150. We note that, for this analysis, any tag insertion/deletion or change in tag composition (e.g., an attribute value) is treated as a new page structure. Consequently, despite the large number of potential HTML structures, both pages are highly amenable to large static templates.

3.4 Subtleties

Handling different template versions: Since Fawkes clients cache static HTML templates, and Fawkes servers can decide to generate new templates based on page modifications or popularity changes, it is possible that different clients have different template versions cached. One option is to have clients check for updates prior to evaluating cached templates using the If-Modified-Since HTTP header, but this would eliminate most of Fawkes's warm cache benefits as a browser would have to incur multiple round trips before rendering any content for the user. Instead, to handle these differences, static templates include a hash of the template contents as a variable in the inline patcher code. The patcher includes this information in its XHR request for a dynamic patch file; no browser modifications are required.

In order to make use of hash information in client requests, Fawkes servers must maintain a mapping of hashes to past static HTML template files which covers the max duration over which the templates are cacheable, i.e., if the templates for a URL are set to be cacheable for 1 day, the Fawkes server must store an entry for each template version served over the past day. Importantly, we expect these storage overheads to be low as our results highlight that templates remain largely unchanged on the order of weeks, and across personalized versions of pages for different users (§2 and §5).

Updating cached templates: Fawkes can use the hash-based approach described above to ensure benefits despite variations in cached templates. However, over time, Fawkes servers may wish to update cached templates to reflect significant changes in page structure that may deem past versions poor in terms of performance. For this, Fawkes servers simply send updated templates along with dynamic patches served with HTTP/2 server push. Because the pushed templates will remain cacheable for longer durations than the currently cached versions, default browsers will automatically replace the cached template for subsequent loads.

Static templates across URLs: In scenarios where static templates are generated for individual URLs by considering their possible HTML variants, templates can be cached directly under the page's URL. However, as we discuss in §2 and §5, Fawkes's template caching approach can provide significant benefits across different URLs of the same page type, e.g., different search result pages or news articles. To support such scenarios efficiently, browsers must slightly alter their caching approach to allow objects to be cacheable across multiple URLs. Websites can specify a regular expression that precisely covers the URLs for which the template applies, and browsers would use the same cached template for any load which matches that regular expression.

4 Implementation

On the server-side, Fawkes's template and dynamic patch generation code are written in 1912 and 462 lines of Python and C++ code, respectively. Both components are implemented as standalone modules for seamless integration with existing web servers and content management platforms [17, 73]. Module inputs are a set of HTML files, and outputs are full formed HTML and JSON files that can be directly shipped to client browsers. For template generation, Fawkes extended the APTED tree comparison tool [55, 56]. HTML parsing and modification are done using Beautiful Soup [60].

On the client-side, Fawkes's JavaScript patcher library consumes 3 KB when compressed with Brotli [27]. The patcher is written entirely using native DOM and JavaScript methods, and is thus compatible with unmodified web browsers. We note that the DOM shims are shared across pages, and thus could be cached as a separate object from each page's static template to reduce bandwidth costs.

5 Evaluation

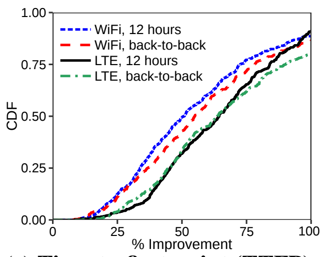
5.1 Methodology

We evaluated Fawkes using two phones, a Nexus 6 (Android Nougat; 2.7 GHz quad core processor; 3 GB RAM) and a Galaxy Note 8 (Android Oreo, 2.4 Ghz octa core; 6 GB RAM). Fawkes performed similarly across the two devices, so we only report results for the Nexus 6. Unless otherwise noted, page loads were run with Google Chrome (v75).

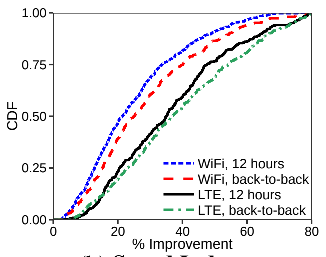
Our experiments consider two different sets of pages:

- Alexa top 500 US landing pages [5]. We augment this list with 100 interior pages that were randomly chosen from a pool of 1000 pages generated by a monkey crawler [3] that clicked links on each site's landing page.
- a smaller set of 20 pages that includes pairs of different pages of the same type. Starting from the Alexa top 50 list, we identified page types that have many versions, and manually generated pairs for each one, e.g., two Google search results and two public Twitter profile pages.

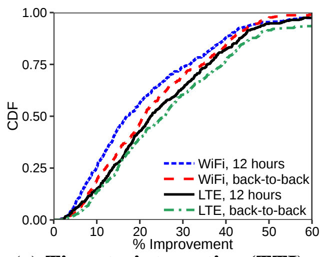
In order to create a reproducible test environment and because Fawkes involves page modifications, our evaluation uses the Mahimahi web record-and-replay tool [48]. We recorded versions of each page in our corpus at multiple times to mimic different warm cache scenarios: back to back page loads, and page loads separated by 12 and 24 hours. Mobile-optimized (including AMP [24]) pages were used when available. To replay pages, we hosted the Mahimahi replay environment on a desktop machine. Mobile phones were connected to the desktop machine via both USB tethering and live wireless networks (Verizon LTE and WiFi) with strong signal strength. The desktop initiated page loads on the mobile device using Lighthouse [28], and all control traffic for this was sent over the USB connection. All web and DNS traffic were sent over the live wireless networks into Mahimahi's replay environment. We modified



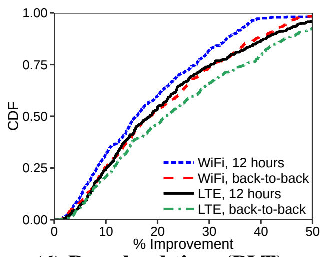
(a) **Time-to-first-paint (TTFP)**



(b) **Speed Index**



(c) **Time-to-interactive (TTI)**



(d) **Page load time (PLT)**

Figure 7: **Distributions of warm cache (back to back and 12 hour) per-page improvements with Fawkes vs. a default browser (i.e., using each page's default HTML) for 600 pages.**

Mahimahi to faithfully replay the use of HTTP/2 (including server push decisions) and server processing delays observed during recording; details of these modifications are listed in §A.1.

In accordance with §3, in all experiments, Fawkes's templates are generated a priori (i.e., offline). We apply server processing delays for a given template as the median delay observed for objects marked as cacheable in the default load of the page; these objects likely represent premade content. Note that this strategy ensures that templates experience the observed server-side delays that do not relate to content generation, e.g., delays due to high server load. Unless otherwise noted, templates are generated using the first and current versions of a page (i.e., a version at time 0, and the version in the back to back load, 12 hours later, etc.); we present results for other template generation strategies in §5.5. Dynamic patches are generated online by Mahimahi's web servers. Server processing times for patches include both the observed server processing time for the page's original HTML file, as well as the time taken to generate a patch.

We evaluate Fawkes on multiple web performance metrics. Page load time was measured as the time between the `navigationStart` and `onload` JavaScript events. We also consider three state-of-the-art metrics which better relate to user-perceived performance: 1) Speed Index (SI), which represents the time needed to fully render the pixels in the intial view of the page, 2) Time-to-first-contentful-paint (TTFP), which measures the time until the first DOM content is rendered to the screen, and 3) Time-to-interactive (TTI), which measures how quickly a page becomes interactive with rendered content, an idle network, and the ability to immediately support user inputs. All three metrics were measured using pwmetrics [32]. In all experiments, we load each page three times with each system under test, rotating amongst them in round robin fashion; we report numbers per system based on the load with the median page load time.

Correctness and limitations: To ensure a faithful evaluation, we analyzed the pages in our 600-page corpus to identify and exclude those that experience replay errors due to either Mahimahi's (22 pages) or Fawkes's limitations (17 pages). Details about our correctness checks are in §A.2.

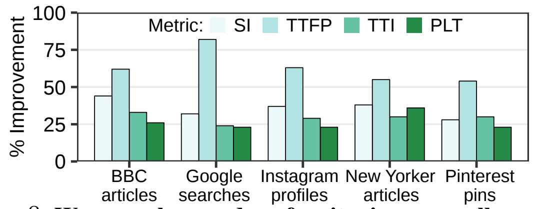


Figure 8: **Warm cache speedups for sites in our smaller corpus.**

5.2 Improving User-Perceived Web Performance

Warm Cache: Figure 7 illustrates Fawkes's ability to improve performance for our 600-page corpus, compared to a default browser, across a variety of web performance metrics and warm cache settings; we omit results for the 24 hour setting due to space constraints, but note that the trends were the same. Benefits with Fawkes are most pronounced on the metrics that evaluate visual loading progress, SI and TTFP. For example, in the 12 hour warm cache setting, median SI improvements are 38% and 22% on the LTE and WiFi networks, respectively. Improvements jump to 67% and 51% for TTFP; these benefits directly characterize Fawkes's immediate rendering of static HTML templates, compared to the lengthy blank screen in a default page load. Table 3 lists the raw time savings pertaining to these improvements.

Despite targeting quick visual feedback, Fawkes's results are also significant for more general web performance metrics like TTI and PLT: median improvements in the 24 hour scenario are 20% and 17% in the LTE setting. The reason is that in warm cache settings, Fawkes enables browsers to utilize network and CPU resources that go idle in standard page loads as HTML objects are being loaded. Browsers can immediately perform required rendering and processing tasks (which are non-negligible on mobile devices [43, 41, 69, 61]) of both template content and referenced cached objects; at the same time, browsers can issue requests for any referenced uncacheable objects to make use of the idle network.

Across all metrics, Fawkes's benefits are higher in LTE settings than on WiFi networks. The reason is that network latencies are higher on LTE networks: in our setup, last mile (access link) RTT values were consistently around 82 ms for LTE and 17 ms for WiFi. Higher round trip times increase the time that default page loads are blocked on fetching top-level HTML objects while Fawkes parses its templates.

As expected, benefits were consistently higher in the back-

Property	back to back	12 hours	24 hours
Static template size (KB)	102 (601)	77 (343)	73 (358)
Dynamic patch size (KB)	6 (249)	44 (491)	52 (460)

Table 1: **Analysis of Fawkes's templates and dynamic patches across warm cache scenarios with different time windows. Results list median (95th percentile) values for each property.**

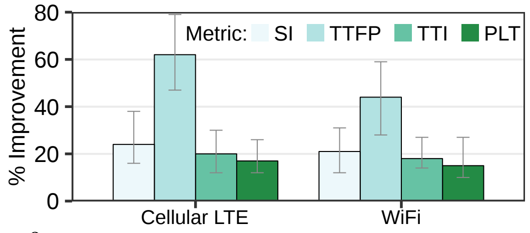


Figure 9: **Cold cache speedups with Fawkes versus a default browser on our 600-page corpus. Bars represent medians, and error bars span from the 25th to 75th percentile.**

to-back warm cache setting than when page loads were separated by 12 or 24 hours. This is because HTML objects undergo fewer changes across back-to-back loads, leading to larger templates and fewer updates (Table 1). Larger templates result in immediate feedback that more closely resembles the final page, as well as increased opportunities to utilize idle CPU and network resources. Note that patch sizes include page content (e.g., inline scripts) to be inserted.

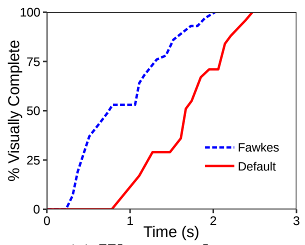
Figure 8 illustrates similar warm cache benefits (over the LTE network) for representative sites in our smaller corpus. Templates are made by consideringdifferent versions of the same page type. We note that TTFP benefits were highest for Google search pages because those pages incur the highest server processing times (for result generation).

Cold Cache: Although Fawkes's template-based approach primarily targets warm cache settings, benefits are significant in cold cache scenarios (Figure 9). For example, median SI and TTFP improvements were 24% and 62% for the LTE network. These results consider templates generated using HTML files generated 24 hours apart. The reason for these benefits mirror those in warm cache settings, but with smaller savings. Since static HTML templates are served faster than dynamic patches, browsers still have a window to perform template rendering and compute tasks with Fawkes while the default page load is blocked. Browsers largely use this time to quickly fetch referenced uncached objects, making better use of the idle network. Like in warm cache settings, SI and TTFP benefits drop to 21% and 44% on the WiFi network due to the decreased network round trip times.

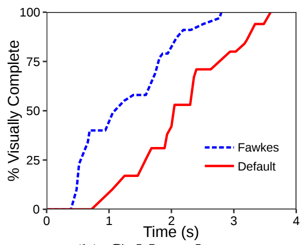
5.3 Understanding Fawkes's Benefits

Case study: To better understand Fawkes's performance, we analyzed the visual progress of page loads both with and without Fawkes. Visual progress tracks the fraction of the browser viewport (i.e., the part of the page that is visible without scrolling) that has been rendered to its final form.

Figure 10 shows warm and cold cache results for a representative site in our corpus, the Yahoo homepage. In the



(a) **Warm cache.**



(b) **Cold cache.**

Figure 10: **Visual progress with and without Fawkes for the Yahoo homepage. Warm cache loads were 12 hours apart.**

warm cache scenario, Fawkes is able to make an immediate jump (53% in 790 ms) in visual progress by parsing and rendering a large part of the static HTML template, as well as referenced static objects which are also in the browser cache. In contrast, the default browser is blocked on the multiple network round trips and server processing delays required to fetch the page's top-level HTML object; visual progress does not increase until 1110 ms into the load. The initial render (29% in 1270 ms) is also much smaller than with Fawkes because the default HTML parse gets quickly blocked on fetching an uncacheable JavaScript file—rendering is blocked until this file is fetched and evaluated. Fawkes also has to fetch this file, but this occurs via an applied update, at which point Fawkes has already reached 53% visual completeness. We note that evaluation of this script (and thus blocked rendering) is 27 ms worse with Fawkes due to overheads from DOM shims. However, these overheads are overshadowed by the large early lead in visual progress that Fawkes achieves.

In the cold cache setting, both Fawkes and the default browser incur network delays to fetch an HTML object for the page. However, this delay is lower for Fawkes as the static template is pre-generated. From this point, the page loads are largely similar to the warm cache setting: Fawkes makes a larger immediate jump in visual progress (40% in 690 ms vs. 17% in 1250 ms) as the default browser gets quickly blocked on fetching an external script, while Fawkes does so only after the template is parsed. From there, both page loads progressively render content, but Fawkes never relinquishes its lead. We note that, though it is not visible in the graphs, Fawkes issues requests for non-blocking external objects (e.g., images) that are listed in the template earlier.

Template content: Fawkes's early template parsing enables browsers to 1) process referenced cached objects sooner in warm cache loads, and 2) quickly issue requests for referenced external objects in cold cache loads. To understand how often these optimizations are applied, we analyzed the static URLs listed in Fawkes's templates; we considered templates generated using loads 12 hours apart. We found that, on the median page, templates referenced 46% of the page's objects, of which 72% were cacheable.

Patch generation: As described in §3, Fawkes opts to run a tree comparison heuristic rather than a state-of-the-art tree diffing algorithm. Fawkes's heuristic is designed to trade off

Algorithm	# of operations	Execution time (ms)
Fawkes's heuristic	2 (667)	20 (59)
Insert-first heuristic	2 (3013)	30 (70)
Delete-first heuristic	2 (3065)	30 (70)
State-of-the-art tree diffing algorithm (§3)	17 (136)	2717 (19702)

Table 2: **Fawkes's dynamic patch generation heuristic yields a desirable tradeoff between patch generation time and patch optimality, compared to other heuristics and a state-of-the-art tree diffing algorithm (which Fawkes uses for template generation). Results list median (95th percentile) values.**

System	SI	TTFP	TTI	PLT
Default	2.9 (3.9)	0.5 (0.5)	3.6 (4.4)	4.0 (5.2)
Fawkes	1.8 (2.9)	0.2 (0.3)	2.8 (3.5)	3.3 (4.2)
Vroom	2.4 (3.4)	0.6 (0.5)	2.9 (3.3)	3.2 (3.8)
WatchTower	2.0 (2.5)	0.6 (0.6)	2.6 (3.0)	2.8 (3.6)
Fawkes + Vroom	1.8 (2.9)	0.2 (0.3)	2.6 (3.14)	2.5 (3.4)

Table 3: **Median warm (cold) cache raw times for our 600-page corpus on an LTE cellular network. All results are in seconds, and warm cache loads are spread by 12 hours.**

patch generation time for optimality (in terms of number of operations). To evaluate Fawkes's heuristic on this tradeoff, we compare it to the tree diffing algorithm Fawkes uses for template generation, and two additional heuristics: 'insert-first' and 'delete-first' breadth-first-search approaches where discrepancies discovered when comparing a level in the template and target are handled by first inserting the missing node or deleting the mismatched node, respectively, and then accounting for any remaining deltas (Table 2). As shown, Fawkes's heuristic runs 2 orders of magnitude faster than existing tree diffing algorithms. Median operations are lower with Fawkes's heuristic due to its *move* operation. 95th percentile operations are 5× worse with Fawkes's heuristic due to the inefficiences described in §3.1, but we note that this large gap is only present in 8% of pages.

Importantly, across all warm cache page loads, Fawkes completes heuristic execution and shipping patches to clients *before* client-side template processing completes; shipping patches before this does not improve performance a template parsing must conclude prior to patch application.

5.4 Comparison with Vroom and WatchTower

We compared Fawkes with two recent mobile web optimization systems, Vroom [61] and WatchTower [47]. With Vroom, web servers user HTTP/2 server push to proactively send static resources that they own to clients ahead of future requests. In addition, Vroom servers send HTTP preload headers [68] to let clients quickly download resources that they will soon need from other domains. In contrast, WatchTower accelerates page loads by selectively using proxy servers based on page structural properties and network conditions. When enabled, a proxy loads a page locally using a headless browser and fast network links, and streams individual resources back to the client for processing. Our evaluation considers WatchTower's HTTPS-sharding mode, where each HTTPS origin runs their own proxy to preserve HTTPS security. Proxies were run on EC2 in California where the WatchTower paper reported the highest speedups.

Table 3 compares Fawkes with Vroom and WatchTower for both cold and warm cache loads of our 600 page corpus over an LTE network; trends were similar on the WiFi network. As shown, Fawkes is able to significantly improve performance on the interactivity-focused metrics compared to these systems. For example, median warm cache Speed Index values were 24% and 10% lower with Fawkes than with Vroom and WatchTower, respectively. Fawkes's TTFP benefits over these systems were 69%-73% since acceleration techniques with WatchTower and Vroom only take affect *after* incurring multiple network round trips and server processing delays to download HTML objects.

Our results also show that Vroom and WatchTower are more effective than Fawkes at reducing PLT; median benefits are 3.3% and 16.6%, respectively. The reason is that both Vroom and WatchTower can mask network round trips required to fetch external objects throughout the page load, including those triggered by non-HTML objects. Fawkes, on the other hand, focuses on early parts of a page load–indeed, targeting startup bottlenecks is what differentiates Fawkes from prior acceleration techniques. Importantly, we note that Fawkes's early-stage optimizations are largely complementary to prior techniques.To validate this, we reran the experiment above using a combination of Fawkes and Vroom. Vroom's server hints on the top-level HTML were sent along with Fawkes's dynamic patches. As shown in Table 3, this combination outperforms any tested system in isolation.

5.5 Additional Results

Stale HTML templates: Our warm cache evaluation considered static templates that were generated using the HTML object at time 0 and the one for the current time (e.g., 12 hours). Although this is possible using the techniques presented in §3.3 to generate representative HTML files for a page, it is not the sole practical deployment scenario. An alternative approach would be to generate static HTML templates with only back-to-back loads at a time 0, and use this for future warm cache loads. To evaluate the impact of such stale templates, we loaded the pages in our corpus using both stale (i.e., generated at time 0) and up-to-date (generated using HTML files at time 0 and the current time) templates. We considered staleness of 12 and 24 hours, and observed minimal performance degradations. For example, on the LTE network, median SI values dropped by only 4.2% and 6.8% for the 12 and 24 hour scenarios; TTFP values were unchanged.

Personalized pages: We selected 20 sites from our 600 page corpus that supported user accounts. For each site, we made two user accounts, selecting different preferences when possible, e.g., order results based on time or popularity. We then generated static templates from the HTML objects that each user account fetched. Finally, we loaded one of the user's pages 12 hours later with a warm cache, and compared performance to that of a default browser. Fawkes was able to reduce SI by 27% and 18% on the LTE and WiFi networks, respectively. It is important to note that these trends may not hold for all personalization strategies. For example, pages like Facebook can display structurally diverse content over time and across users. However, our results illustrate that many pages do remain structurally similar across users.

Energy savings and other browsers: Fawkes reduces (per-page) energy usage by 7-18%, and its speedups persist across other browsers (e.g., Firefox). Details provided in §A.3.

6 Related Work

Server push systems: Numerous systems, including Vroom (§5.4), aim to accelerate mobile page loads by leveraging HTTP/2's server push feature [9], where servers proactively push resources to clients in anticipation of future requests [61, 19, 70]. Fawkes is largely complementary to server push systems as these approaches reduce fetch times for resources loaded after the top-level HTML. In contrast, Fawkes speeds up page startup times.

Proxy and backend accelerators: Compression proxies [4, 63, 67, 52] compress objects in-flight between clients and servers, while remote dependency resolution proxies [65, 47, 48, 64] perform object fetches on behalf of clients. Fawkes is orthogonal to these approaches, and can mask the network indirection and computation overheads associated with proxying. In addition, Fawkes preserves the end-to-end nature of the web, avoiding the security challenges of proxying.

More recently, Prophecy [43], Shandian [71], and Opera Mini [51] return post-processed versions of objects to reduce client-side computation and bandwidth costs. All three systems must incur the same network round trips and (more) server processing delays that default page loads to download top-level HTML objects–only then do their acceleration techniques help. These delays are exactly what Fawkes aims to alleviate. We also note that Fawkes's patcher and shims tackle a fundamentally different challenge than those in Prophecy and Shandian: Fawkes must execute JavaScript code in an environment with fast-forwarded DOM state.

Dependency-aware scheduling: Certain systems have improved the scheduling of network requests based on inherent dependencies in page content. Klotski [14] analyzes pages offline to identify high-priority objects in terms of user utility, and uses knowledge of network bandwidth to stream them to clients before they are needed. Polaris [42] uses a client-side request scheduler that reorders requests to minimize the number of effective round trips in a page load without violating state dependencies. However, both systems are unable to process or render content prior to an HTML download. Thus, these systems can work side by side with Fawkes.

JavaScript UI frameworks: Libraries like Vue.js [74], AngularJS [26], and React [22] efficiently update client-side page state during page loads. A key feature across these frameworks is the use of a virtual DOM, where JavaScript-based DOM updates are first performed on a lightweight DOM representation, and aggregate results (rather than intermediate layout and render events) are applied to the actual DOM. Using such efficient update strategies, these frameworks support client-side page rendering, whereby a page's top-level HTML embeds only a single JavaScript library that is responsible for downloading and rendering downstream page content. While these frameworks focus on efficiently updating content during a page load and require developers to rewrite pages, Fawkes operates on unmodified pages and aims to quickly display content shared across page loads. Further, unlike the client-side page rendering approach, Fawkes's static templates embed both the JavaScript patcher library *and* all of a page's static HTML content. This, in turn, ensures that Fawkes can render static content while fetching downstream (dynamic) content.

Accelerating HTML loading: Google's SDCH [15] allows web servers to specify cacheable components of HTML files; on subsequent loads, servers need only send new components or deltas to cached ones, thereby saving bandwidth. Unlike Fawkes, SDCH does not allow browsers to render cached HTML components *until the entire HTML is constructed*. Thus, SDCH does not face the view invariance challenges that Fawkes's patcher does, and SDCH is unable to reduce web startup times by rendering cached HTML content quickly for users. Other industry efforts have focused on dividing pages into modular components called "pagelets", which can be generated and processed independently and in parallel [13, 33]. Pagelets share Fawkes's goal of improving resource utilization to more quickly display content to users. However, unlike Fawkes, individual pagelets do not include a mechanism for automatically separating static and dynamic HTML, and instead use a single response that is shipped only after the pagelet's dynamic content is generated.

7 Conclusion

Inspired by the mobile app startup process, this paper presents Fawkes, a mobile web acceleration system that generates cacheable, static HTML templates that can be immediately rendered to quickly display content to users as page updates are fetched. Fawkes represents a shift in the web acceleration space, by focusing on leveraging underutilized resources at the *beginning* of page loads. We find that Fawkes brings median warm cache reductions of 46%, 64%, 26%, and 22% for SI, TTFP, TTI, and PLT, and outperforms state-of-the-art server push and proxy-based acceleration systems by 10%-24% and 69%-73% on SI and TTFP.

Acknowledgements: We thank Harsha Madhyastha and James Mickens for their valuable feedback on earlier drafts of the paper, as well as Ben Greenstein and Michael Buettner for useful discussions on Fawkes' practical implications. We also thank our shepherd, Ankit Singla, and the anonymous NSDI reviewers for their constructive comments.

REFERENCES

[1] Overleaf: A web-based collaborative latex editor. https://github.com/overleaf/overleaf.

[2] Reddit. https://github.com/reddit-archive/reddit.

[3] Seleniumhq browser automation. https://selenium.dev/, 2019.

[4] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. NSDI '15. USENIX, 2015.

[5] Alexa. Top Sites in the United States. http://www.alexa.com/topsites/countries/US, 2018.

[6] Amazon. Silk Web Browser. https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html, 2018.

[7] D. An. Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/, 2018.

[8] APOWERSOFT. Apowersoft Screen Recorder. https://play.google.com/store/apps/details?id=com.apowersoft.screenrecord&hl=en_US, 2019.

[9] M. Belshe, R. Peon, and M. Thomson. HTTP/2.0 Draft Specifications. https://http2.github.io/, 2018.

[10] D. Bhattacherjee, M. Tirmazi, and A. Singla. A Cloud-based Content Gathering Network. In *Proceedings of HotCloud*, 2017.

[11] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-perceived Quality into Web Server Design. World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Networking, 2000.

[12] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. CHI, The Hague, The Netherlands, 2000. ACM.

[13] A. Brousseau. Generating Web Pages in Parallel with Pagelets, the Building Blocks of Yelp.com. https://engineeringblog.yelp.com/2017/07/generating-web-pages-in-parallel-with-pagelets.html, 2017.

[14] M. Butkiewicz, D. Wang, Z. Wu, H. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2015.

[15] J. Butler, W.-H. Lee, B. McQuade, and K. Mixter.

[16] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. In *Proceedings of the 34th International Conference on Automata, Languages and Programming*, ICALP'07, pages 146–157, Berlin, Heidelberg, 2007. Springer-Verlag.

[17] Drupal. Drupal - Open Source CMS. https://www.drupal.org/, 2019.

[18] E. Enge. MOBILE VS. DESKTOP USAGE IN 2019. https://www.perficientdigital.com/insights/our-research/mobile-vs-desktop-usage-study, 2019.

[19] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan. Towards a SPDY'Ier Mobile Web? *IEEE/ACM Trans. Netw.*, 23(6):2010–2023, Dec. 2015.

[20] D. Etherington. Mobile internet use passes desktop for the first time, study finds. https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds/, 2016.

[21] T. Everts and T. Kadlec. WPO stats. https://wpostats.com/, 2019.

[22] Facebook. React: A JavaScript library for building user interfaces. https://reactjs.org/, 2019.

[23] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web Site Delays: How Tolerant are Users? Journal of the Association for Information Systems, 2004.

[24] Google. Accelerated Mobile Pages Project – AMP. https://www.ampproject.org/.

[25] Google. Speed Index - WebPagetest Documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2012.

[26] Google. AngularJS: Superheroic JavaScript MVW Framework. https://angularjs.org/, 2019.

[27] Google. Brotli compression format. https://github.com/google/brotli, 2019.

[28] Google. Lighthouse. https://developers.google.com/web/tools/lighthouse/, 2019.

[29] Google. Progressive Web Apps. https://developers.google.com/web/progressive-web-apps/, 2019.

[30] Google. Service Workers: an Introduction. https://developers.google.com/web/fundamentals/primers/service-workers/, 2019.

[31] Google Android. Understanding Systrace. https://source.android.com/devices/tech/debug/systrace, 2019.

[32] P. Irish. pwmetrics: Progressive web metrics. https://github.com/paulirish/pwmetrics, 2019.

[33] C. Jiang. BigPipe: Pipelining web pages for high performance. https://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919/, 2010.

[34] B. Jun, F. E. Bustamante, S. Y. Whang, and Z. S. Bischof. AMP up your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2019.

[35] C. Kelton, J. Ryoo, A. Balasubramanian, and S. R. Das. Improving User Perceived Page Load Time Using Gaze. In *Proceedings of the 14th USENIX Confer-*

ence on Networked Systems Design and Implementation, NSDI'17, pages 545–559. USENIX Association, 2017.

[36] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, pages 91–102, London, UK, UK, 1998. Springer-Verlag.

[37] G. Li, E. Andreasen, and I. Ghosh. Symjs: Automatic symbolic testing of javascript web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 449–459, New York, NY, USA, 2014. ACM.

[38] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant Web Browsing for Mobile Devices. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII. ACM, 2012.

[39] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 11. USENIX Association, 2010.

[40] Monsoon Solutions Inc. Power monitor software. http://msoon.github.io/powermonitor/, 2018.

[41] J. Nejati and A. Balasubramanian. An In-depth Study of Mobile Browser Performance. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1305–1315. International World Wide Web Conferences Steering Committee, 2016.

[42] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX, 2016.

[43] R. Netravali and J. Mickens. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2018. USENIX Association.

[44] R. Netravali and J. Mickens. Remote-Control Caching: Proxy-based URL Rewriting to Decrease Mobile Browsing Bandwidth. In *Proceedings of the 19th International Workshop on Mobile Computing Systems and Applications*, HotMobile '18. ACM, 2018.

[45] R. Netravali and J. Mickens. Reverb: Speculative Debugging for Web Applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 428–440, New York, NY, USA, 2019. Association for Computing Machinery.

[46] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring Time-to-Interactivity for Web Pages. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX, 2018.

[47] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 430–443. ACM, 2019.

[48] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. Proceedings of ATC '15. USENIX, 2015.

[49] J. Newman and F. Bustamante. The Value of First Impressions: The Impact of Ad-Blocking on Web QoE". In D. Choffnes and M. Barcellos, editors, *Passive and Active Measurement - 20th International Conference, PAM 2019, Proceedings*, pages 273–285, Germany, 1 2019. Springer Verlag.

[50] NodeJS. Express: Fast, unopinionated, minimalist web framework for Node.js. https://expressjs.com/, 2019.

[51] Opera. Opera Mini. http://www.opera.com/mobile/mini, 2018.

[52] Opera. Opera Turbo. http://www.opera.com/turbo, 2018.

[53] V. N. Padmanabhan and J. C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996.

[54] M. Pawlik and N. Augsten. Rted: A robust algorithm for the tree edit distance. *Proc. VLDB Endow.*, 5(4):334–345, Dec. 2011.

[55] M. Pawlik and N. Augsten. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, Mar. 2015.

[56] M. Pawlik and N. Augsten. APTED algorithm for the Tree Edit Distance. https://github.com/DatabaseGroup/apted, 2018.

[57] C. Petrov. 52 Mobile vs. Desktop Usage Statistics For 2019 [Mobile's Overtaking!]. https://techjury.net/stats-about/mobile-vs-desktop-usage/, 2019.

[58] Pylons Project. Pylons Project. https://pylonsproject.org/, 2019.

[59] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Give in to Procrastination and Stop Prefetching. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII. ACM, 2013.

[60] L. Richardson. Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2019.

[61] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM. ACM,

2017.

[62] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[63] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual Conference on Mobile Computing and Networking*, MobiCom. ACM, 2015.

[64] A. Sivakumar, C. Jiang, S. Nam, P. Shankaranarayanan, V. Gopalakrishnan, S. Rao, S. Sen, M. Thottethodi, and T. Vijaykumar. Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, Mobicom. ACM, 2017.

[65] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 325–336, New York, NY, USA, 2014. ACM.

[66] M. Varvello, J. Blackburn, D. Naylor, and K. Papagiannaki. EYEORG: A Platform For Crowdsourcing Web Quality Of Experience Measurements. In *Proceedings of the 12th Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16. ACM, 2016.

[67] J. Volpe. Nokia Xpress brings cloud-based compression to the Lumia line. Engadget. https://www.engadget.com/2012/10/03/nokia-xpress-brings-cloud-based-compression-to-the-lumia-line/, October 3, 2012.

[68] W3C. Preload. Editor's Draft. https://w3c.github.io/preload/, January 9, 2018.

[69] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[70] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proceedings of NSDI*, NSDI'14, pages 387–399, Berkeley, CA, USA, 2014. USENIX Association.

[71] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.

[72] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12. ACM, 2012.

[73] WordPress. Blog Tool, Publishing Platform, and CMS – WordPress. https://wordpress.org/, 2019.

[74] E. You. Vue.js: The Progressive JavaScript Framework. https://vuejs.org/, 2019.

[75] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, Dec. 1989.

[76] T. Zimmermann, B. Wolters, O. Hohlfeld, and K. Wehrle. Is the Web ready for HTTP/2 Server Push? In *Proceedings of the 14th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT. ACM, 2018.

A Appendix

A.1 Mahimahi Modifications

To compute per-object server processing delays, we first recorded the RTT to each origin in a page as the median time between the TCP SYN and SYN/ACK packets across all connections with that origin. We then defined the server processing delay for an object as its TTFB minus 1 RTT (for the transmission of the HTTP request and initial response bytes); when applicable, we also subtracted out connection setup delays (1 or 2 RTTs depending on whether the resource was downloaded via HTTP or HTTPS). Lastly, we modified Mahimahi's `replayserver` to wait for the corresponding server processing delay before shipping back any object.

A.2 Correctness and Limitations

To ensure a faithful evaluation of Fawkes, we analyzed the pages in our corpora to identify and exclude pages that experience replay errors due to either Mahimahi or Fawkes. We excluded 22 pages due to Mahimahi replay errors, most of which were the result of SSL errors for pages that leverage the Server Name Indication (SNI) feature in SSL/TLS certificates (which Mahimahi does not support), and missing resources that Mahimahi's URL matching heuristic was unable to remedy.

On the remaining pages, correctness with Fawkes was evaluated by forcing determinism upon JavaScript execution (e.g., using fixed return values for calls to `Math.Random()`) [39, 45], and comparing loads with and without Fawkes in three ways: 1) a pixel-by-pixel analysis of the final page (using pwmetrics' screenshots and visual analysis tools [32]), 2) the number of registered JavaScript event handlers (logged using shims on the `addEventListener` mechanism and by iterating over the DOM tree after the `onload` event fired [46]), and 3) the browser console errors printed during the page load. We excluded the 17 pages that differed on any of these three properties from our evaluation. Further investigation revealed two key reasons for such discrepancies, which are limitations with Fawkes:

- Although Fawkes cuts downstream JavaScript in templates after the first tag removal or alteration, it does not remove CSS. The reason is that CSS rules can significantly affect the styling of template content, bringing it closer to the final page version. However, CSS and JavaScript code can share state in the form of DOM node attributes. As a result, downstream CSS files in a page's template can modify DOM attribute state that patched (upstream) JavaScript code can subsequently access—this can alter page execution and lead to errors.
- JavaScript code can dynamically rewrite downstream HTML using the `document.write()` interface. However, Fawkes's patches are based on a page's static HTML, which does not reflect JavaScript execution. Thus, because our current implementation of Fawkes does not use shims for `document.write()`, it is possible for JavaScript code in the template to (correctly) rewrite downstream HTML content, that is (incorrectly) resurrected by the Fawkes patcher.

A.3 Additional Results

Energy consumption: To examine the impact that Fawkes has on energy usage, we connected a Nexus 6 phone to a Monsoon power monitor [40] and loaded our 600 page corpus. During cold cache loads, Fawkes's speedups reduce median per-page energy usage by 11% and 7% compared to a default browser on the LTE and WiFi networks, respectively. Benefits jump to 18% and 11% in warm cache settings (12 hours apart). In both cases, benefits are higher on LTE due to the higher network latencies and the fact that LTE radios consume more energy than WiFi hardware when active [65].

Additional browsers: Since Fawkes does not require any browser modifications, we also evaluated Fawkes with Firefox (v68) using our 600 page corpus and the same experimental setup from §5.1. Benefits in the 12 hour warm cache setting were quite comparable, despite Firefox using a different rendering engine than Chrome. Fawkes reduced median SI by 21% and 34% on the WiFi and LTE networks.

A.4 Additional Related Work

Mobile-optimized pages: Certain sites cater to mobile settings by serving pages that involve less client-side computation, fewer bytes, and fewer network fetches. For example, Google AMP [24, 34] is a recent mobile web standard that requires developers to rewrite pages using restricted forms of HTML, JavaScript, and CSS. Unlike AMP, Fawkes accelerates legacy pages without needing developer effort. Further, Fawkes provides complementary benefits and can lower AMP startup delays: Fawkes's TTFP and SI reductions were 58% and 27% for the 23 AMP pages in our corpus.

Prefetching: Prefetching systems predict user browsing behavior and optimistically download objects prior to user page loads [53, 38, 72]. Unfortunately, such systems have witnessed minimal adoption due to challenges in predicting what pages a user will load and when; inaccurate page and timing predictions can waste device resources or result in stale page content [59]. By rendering static templates as soon as a user navigates to a page, Fawkes is able to achieve comparable TTFP reductions without the issues of prefetching.

Progressive Web Apps (PWAs): Google recently proposed PWAs [29], applications that are written using standard web languages (e.g., HTML, JavaScript), can be loaded by a standard web browser, but are installed as an application on a user device. PWAs use service workers [30] which employ aggressive caching and custom update logic to run offline and support push notifications from servers. Fawkes shares the idea of improving use of web caching and app-like update logic. However, in contrast to PWAs which require developer effort for creation (and potentially maintenance), Fawkes transparently applies app-like templating to legacy pages.

VMscatter: A Versatile MIMO Backscatter

Xin Liu[*], Zicheng Chi[*], Wei Wang, Yao Yao, and Ting Zhu
University of Maryland, Baltimore County

Abstract

In this paper, we design and implement a versatile MIMO backscatter (VMscatter) system, which leverages the diversity features of MIMO to dramatically decrease bit error rate (BER) and increase throughput with negligible overhead. Our approach is different from existing WiFi MIMO backscatter approaches which simply reflect the signals from the WiFi MIMO sender and do not take advantage of MIMO technologies' advanced features (i.e., low bit error rate and high throughput). In our approach, the backscatter can achieve the same full diversity gain as traditional MIMO system by implementing the space-time coding on the backscatter tag under the constraint that backscatter tags cannot control the reflected signals to be orthogonal. Moreover, the backscatter can reflect excitation signals from the senders that have either a single antenna or multiple antennas. To implement the VMscatter system, we addressed the special design challenges such as complicated channel estimations among the sender, tag, and receiver by using a novel pre-scatter channels elimination method and a post-scatter channels equalization method. Our VMscatter design introduces negligible overheads (in terms of hardware cost, energy consumption, and computation) on the backscatter tag. We further extended our design to support any number of antennas that the sender, tag, and receiver have. Our MIMO backscatter design is generic and has the potential to be extended to achieve massive MIMO. We extensively evaluated our system in different real-world scenarios. Results show that the BER is reduced by a factor of 862 compared to the most related work MOXcatter [68].

1 Introduction

In the last few years, backscatter systems have been proposed to piggyback data on ambient signals such as WiFi [33, 59], Bluetooth [30], LoRa [28, 47], FM [53], etc. By doing this, the backscatter device consumes very little energy to wirelessly transmit data, which can enable lots of Internet-of-Things applications, such as device tracking [31], smart homes and smart health [5, 30]. On the other hand, the Multiple-Input Multiple-Output (MIMO) technique has become an essential element of wireless communication. To explore the benefits of the MIMO technique, researchers have proposed various important approaches, such as full duplex MIMO [10, 11, 15], multi-user MIMO [9, 19, 35, 48–50, 56], massive MIMO [12, 29, 32, 65, 66], distributed MIMO [26, 27], and MIMO networks [21, 25, 43, 44, 57, 67].

Although MIMO has been widely used and explored in wireless systems, little work has been conducted to effectively integrate MIMO techniques into a backscatter system for more reliable and faster backscatter communications. The only related work (MOXcatter [68]) tries to backscatter MIMO signals from the WiFi MIMO sender. However, the multiple antennas on MOXcatter do not take advantage of advanced features of MIMO technology. Specifically, although MOXcatter uses multiple antennas, the phase changes among these antennas are always the same when reflecting the WiFi signals. Therefore, MOXcatter does not fully leverage the spatial diversity, which leads to even higher bit error rate and lower throughput than the corresponding non-MIMO WiFi backscatter system FreeRider [62]. Moreover, the MOXcatter tag needs explicit control signals from the sender to identify whether the sender is sending a single stream or multiple streams signals. These explicit control signals prevent MOXcatter to be widely deployed in an environment where the sender is uncontrollable.

Different from existing WiFi backscatters that reflecting non-MIMO [14, 33, 61, 62] and MIMO [68] signals, our high-level goal in this paper is to design a versatile MIMO backscatter (VMscatter) system, which leverages the diversity features of MIMO to dramatically decrease bit error rate (BER) with negligible overhead because the backscatter communication has a very low signal-to-noise ratio (SNR) which leads to a high BER. With a lower BER, the backscatter communication can have a longer effective communication range, better robustness, and better reliability. We note that the sender of VMscatter is not required to be a MIMO device. In our design, we encountered the following challenges:

C1. How to realize MIMO transmissions on a low-power

[*]Both authors contributed equally to the paper

backscatter tag for reliable communication? A traditional MIMO system improves the reliability by utilizing a space-time coding scheme to generate orthogonal symbols (which increase the diversity gain) among antennas. However, the low-power backscatter tag cannot demodulate the incoming ambient signals (which are used to piggyback backscatter data) due to limited resources on the tag. Therefore, the backscatter tag is not able to generate orthogonal symbols. We discovered that by implementing space-time coding on the backscatter tag, we can achieve the same full diversity gain as traditional MIMO systems, even though the reflected backscattered symbols are non-orthogonal. We note that the non-orthogonality is not a hard constraint. Our proposed algorithm is applicable for the reflected symbols being either orthogonal or non-orthogonal. However, it is very challenging to implement space-time coding on the tag by only turning on/off the switches, because we do not want to increase the tag's computation and energy overheads.

C2. How to demodulate the backscattered data? The MIMO backscatter system is more complicated than traditional MIMO system because we need to deal with the pre-scatter channels (i.e., the channels between the sender and the backscatter tag) and post-scatter channels (i.e., the channels between the backscatter tag and the receiver) for proper demodulation. For example, a 2×2 MIMO system has four different physical channels among the two sender antennas and two receiver antennas. The receiver needs to estimate these four channels in order to correctly demodulate the data. With a two-antenna backscatter tag in the middle (i.e., $2 \times 2 \times 2$ setup), the situation becomes even more complicated. To obtain the backscattered data, eight physical channels need to be estimated (i.e., four pre-scatter channels and four post-scatter channels). Furthermore, we also considered frequency mismatch (including cumulative clock drift and oscillator instability) between the VMscatter tag and receiver as well.

C3. How to achieve a generic $M \times K \times N$ setup? As a versatile MIMO backscatter system, it needs to support an arbitrary number of antennas at the sender, tag, and receiver sides (shown in Fig. 1). It is important to support any number (i.e., M) of antennas that the senders may have, because we want to leverage various types of existing infrastructures. However, it is very challenging to achieve this because the lower-power backscatter tag cannot decode the packet from the sender to obtain the knowledge of the number of antennas. Existing work such as MOXcatter [68]) needs to modify the sender to explicitly send out control messages (i.e., four consecutive special packets) to the tag so that the tag can identify the types of the sender's excitation signals (i.e., MIMO v.s. non-MIMO), which is impractical. We mathematically demonstrate how to eliminate the impact of the pre-scatter channels such that the impact of the number (i.e., M) of senders' antennas can be eliminated (see Sec. 4.3). Therefore, we do not need to modify the sender. We further describe how to support an arbitrary number of antennas at both the tag (i.e., K) and receiver (i.e., N) sides (see Sec. 4.4). We also validate our design using both experiments and simulations.

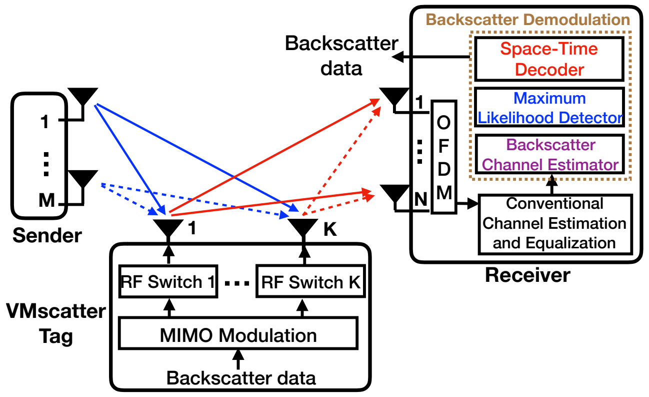


Figure 1: System Architecture

Our key contributions are as follows:

• To the best of our knowledge, this is the first work that designed and implemented a versatile MIMO system, which leverages the diversity features of MIMO to dramatically decrease bit error rate (BER) and increase throughput with negligible hardware cost, energy consumption, and computation overheads on the low-power backscatter tag.

• To implement the versatile MIMO backscatter system, we addressed the special design challenges on the backscatter tag (e.g., how to implement space-time coding by only turning on/off the switches) and on the receiver side (e.g., frequency mismatch and complicated channel estimations among the sender, tag, and receiver).

• Our design is generic. It can support an arbitrary number of antennas at the sender, tag, and receiver. The design principles have the potential to be extended to achieve massive MIMO.

• We build a hardware prototype of the proposed VMscatter system and design an IC circuit. We also extensively evaluated our system under different real-world settings. Evaluation results demonstrate that VMscatter can provide faster and orders of magnitude more reliable communication than state-of-the-art backscatter systems. For example, VMscatter's bit error rate is smaller than that of MOXcatter by a factor of more than 800.

2 Motivation and Design Overview

Our work is motivated by the recent advances in MIMO technology, such as Surface MIMO [16] which enables MIMO communication between devices using surfaces coated with conductive paint or cloth. For example, a single-antenna sender-receiver pair can achieve 3×3 MIMO communication by using two points of contact on the conductive surface. By doing this, the number of antennas on the sender and receiver sides can be as low as 1.

We also note that with the advances in printed antennas [3, 23] and RF Micro-Electro-Mechanical Systems (MEMS) [4, 6, 36] technologies, multiple antennas can be integrated into a device. Therefore, it is essential to explore how to achieve MIMO on backscatter tags and what the benefits are.

Applications. Since MIMO technology can enable reliable and fast communication, by leveraging the advantages of MIMO technology, our VMscatter has the potential to be applied to applications in noisy and multipath-rich environments, such as smart buildings and smart cities for low-power and reliable sensing data collection.

As shown in Fig. 1, our system contains three parts:

Sender: The sender does not need to be controlled for transmitting explicit control messages. The sender can be equipped with either a single antenna or multiple antennas.

VMscatter Tag: The backscatter tag piggybacks backscatter data on the ambient signals emitted by the sender. To support a variety of existing infrastructures, our VMscatter tag can naturally support any number of sender antennas without the need of explicit control messages. The MIMO modulation module not only modulates the binary data into in-phase and quadrature values but also encodes the data depending on the number of available antennas. Two coding methods can be used for reducing bit error rate or increasing throughput (details in Sec. 4.2). The RF switches for different antenna ports reflect the incoming RF signals based on the output of the MIMO modulation module. By doing this, the backscatter data piggybacks on the ambient RF signal and the MIMO signals are transmitted out.

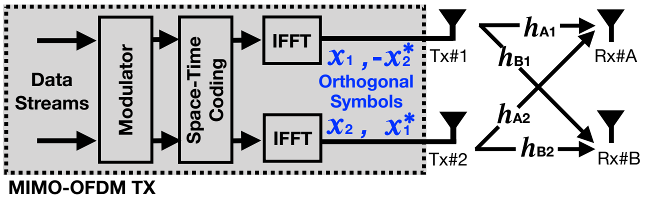
Receiver: The receiver takes the incoming signal and decomposes the signal by using an FFT module. A backscatter packet structure is proposed to cooperate with the conventional channel estimation and equalization module for eliminating the impact of the pre-scatter channel (the channel between the sender and the VMscatter tag). A backscatter demodulation module (including the backscatter channel estimator, maximum likelihood detector, and space-time decoder) is designed on top of the conventional channel estimation and equalization module. The backscatter channel estimator and maximum likelihood detector are used to combat the impact of the post-scatter channel, cumulative clock drift, and oscillator instability. The space-time decoder decodes the data obtained from all the antennas according to the coding method (detailed in Sec. 4.3).

3 Space-Time Coding for MIMO Backscatter

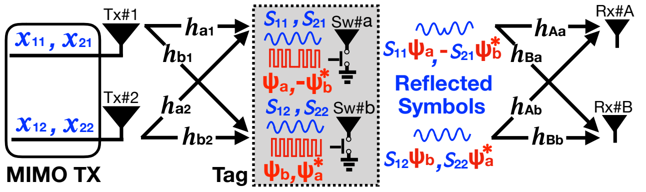
In this section, we first provide the background of a traditional MIMO system. then, we describe why we cannot use existing backscatter systems to achieve MIMO. In the end, we discuss why VMscatter is different.

3.1 Background of Space-Time Coding for MIMO System

We first review the background of space-time coding for the $M \times N$ MIMO-OFDM system which serves as the base of the advanced WiFi protocol (e.g., IEEE 802.11n). For the sake of simplicity, we show the structure of a typical 2×2 MIMO-OFDM system with Alamouti space-time coding [7] in Fig. 2(a). Traditional MIMO systems use complex modules



(a) The structure of a typical 2×2 Alamouti MIMO system



(b) The structure of MIMO backscatter for a $2 \times 2 \times 2$ setup. The pre-scatter and post-scatter channels are $\mathbf{H_R}$ and $\mathbf{H_T}$.

Figure 2: 2×2 Alamouti MIMO system and $2 \times 2 \times 2$ MIMO Backscatter. Conventional MIMO systems need to create the orthogonal symbols to construct the space-time coding. MIMO backscatter can realize the diversity gain of the space-time coding by using the non-orthogonal symbols.

to create orthogonal symbols which provide the full diversity gain [55] for reliable wireless communications. Fig. 2(a) shows the creation process of the orthogonal symbols. First, the data streams (i.e., bit streams) are modulated to complex values (i.e., in-phase and quadrature values). Next, the space-time coding module encodes the complex values based on Alamouti code scheme and feeds the coded data into the Inverse Fast Fourier Transform (IFFT) module for each antenna. Then, each IFFT module efficiently allocates the coded data on multiple subcarriers to form two OFDM symbols (e.g., $\{x_1, -x_2^*\}$ or $\{x_2, x_1^*\}$). Finally, two antennas are used to transmit the four OFDM symbols in two time slots. Because of the orthogonal design in the code scheme, the code matrix formed by the four transmitted symbols fulfills the orthogonal property [54]:

$$\begin{bmatrix} x_1 & x_2 \\ -x_2^* & x_1^* \end{bmatrix}^{\mathrm{H}} \cdot \begin{bmatrix} x_1 & x_2 \\ -x_2^* & x_1^* \end{bmatrix} = (|x_1|^2 + |x_2|^2) \cdot \begin{bmatrix} 1 & \\ & 1 \end{bmatrix} \quad (1)$$

where $(\cdot)^{\mathrm{H}}$ denotes the complex conjugate transpose operator, $(\cdot)^*$ denotes the conjugate operator. Hence, Alamouti code has a full diversity gain [55].

During wireless transmission, the transmitted symbols suffer from channel fading and at the receiver, their sum will be received. Therefore, the 2×2 MIMO system with Alamouti Code can be represented by the following:

$$\begin{aligned} \begin{bmatrix} y_{1A} \\ y_{1B} \end{bmatrix} &= \begin{bmatrix} h_{A1} & h_{A2} \\ h_{B1} & h_{B2} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} n_{1A} \\ n_{1B} \end{bmatrix} \\ \begin{bmatrix} y_{2A} \\ y_{2B} \end{bmatrix} &= \begin{bmatrix} h_{A1} & h_{A2} \\ h_{B1} & h_{B2} \end{bmatrix} \cdot \begin{bmatrix} -x_2^* \\ x_1^* \end{bmatrix} + \begin{bmatrix} n_{2A} \\ n_{2B} \end{bmatrix} \end{aligned} \quad (2)$$

where $[y_{1A}, y_{1B}]^{\mathsf{T}}$ denotes the received vector in the first time slot by receive antennas 1 and 2 respectively, $[y_{2A}, y_{2B}]^{\mathsf{T}}$ denotes the received vector in the second time slot by receive antennas 1 and 2 respectively, h_{nm} denotes the channel impulse response from the m^{th} transmit antenna to the n^{th} receive antenna, and n_{nm} denotes the corresponding noise.

3.2 Why can't we use existing backscatter systems to achieve MIMO?

Since the existing backscatter tag (such as the system in [33, 62, 68]) cannot afford the demodulation (which requires power hungry modules) of incoming signals, the orthogonal property is not guaranteed for backscattered signals. As shown in Fig. 2(b), let us assume the pre-scatter channel (i.e., the channel between the sender and VMscatter tag) is $\mathbf{H_R}$ and the post-scatter channel (i.e., the channel between VMscatter tag and the receiver) is $\mathbf{H_T}$:

$$\mathbf{H_R} = \begin{bmatrix} h_{a1} & h_{a2} \\ h_{b1} & h_{b2} \end{bmatrix}, \quad \mathbf{H_T} = \begin{bmatrix} h_{Aa} & h_{Ab} \\ h_{Ba} & h_{Bb} \end{bmatrix}$$

In the t^{th} time slot, let x_{tk} be the transmitted symbol on the k^{th} transmit antenna, s_{tk} be the incoming symbol on the k^{th} backscatter switch and n^{R}_{tk} be the corresponding noise. Then, the incoming symbol vectors in two time slots can be represented by the following:

$$\begin{bmatrix} s_{1a} \\ s_{1b} \end{bmatrix} = \mathbf{H_R} \cdot \begin{bmatrix} x_{11} \\ x_{12} \end{bmatrix} + \begin{bmatrix} n^{\mathrm{R}}_{1a} \\ n^{\mathrm{R}}_{1b} \end{bmatrix}, \quad \begin{bmatrix} s_{2a} \\ s_{2b} \end{bmatrix} = \mathbf{H_R} \cdot \begin{bmatrix} x_{21} \\ x_{22} \end{bmatrix} + \begin{bmatrix} n^{\mathrm{R}}_{2a} \\ n^{\mathrm{R}}_{2b} \end{bmatrix} \quad (3)$$

From Eqn. 3, we can see that the information of the incoming symbol s_{tk} includes the channel interference $\mathbf{H_R}$ and the baseband value of the transmitted symbol x_{tk}. The backscatter cannot estimate the channel or demodulate the baseband signals without high power components. Therefore, the backscatter does not have knowledge about the incoming signals and cannot guarantee the backscattered signals to be orthogonal.

As a result, even though the backscatter leverages the space-time coding scheme to code its data, the reflected symbols are not orthogonal. For example, as shown in Fig. 2(b), if the backscatter wants to transmit the data $\{\psi_a, \psi_b\}$, the coded data being modulated into the incoming symbols $[s_{1a}, s_{1b}, s_{2a}, s_{2b}]$ are $[\psi_a, \psi_b, -\psi_b^*, \psi_a^*]$. Therefore, the reflected symbols are $[s_{1a}\psi_a, s_{1b}\psi_b, -s_{2a}\psi_b^*, s_{2b}\psi_a^*]$. The representation of the received vectors in two time slots are:

$$\begin{aligned} \begin{bmatrix} y_{1A} \\ y_{1B} \end{bmatrix} &= \mathbf{H_T} \cdot \begin{bmatrix} s_{1a}\psi_a \\ s_{1b}\psi_b \end{bmatrix} + \begin{bmatrix} n^{\mathrm{T}}_{1A} \\ n^{\mathrm{T}}_{1B} \end{bmatrix} \\ \begin{bmatrix} y_{2A} \\ y_{2B} \end{bmatrix} &= \mathbf{H_T} \cdot \begin{bmatrix} -s_{2a}\psi_b^* \\ s_{2b}\psi_a^* \end{bmatrix} + \begin{bmatrix} n^{\mathrm{T}}_{2A} \\ n^{\mathrm{T}}_{2B} \end{bmatrix} \end{aligned} \quad (4)$$

where n^{T}_{tm} is the noise of the received symbol y_{tm}. By plugging the reflected symbols into Eqn. 1, we obtain

$$\begin{bmatrix} s_{1a}\psi_a & s_{1b}\psi_b \\ -s_{2a}\psi_b^* & s_{2b}\psi_a^* \end{bmatrix}^{\mathrm{H}} \cdot \begin{bmatrix} s_{1a}\psi_a & s_{1b}\psi_b \\ -s_{2a}\psi_b^* & s_{2b}\psi_a^* \end{bmatrix} \neq \mathbb{R} \cdot \begin{bmatrix} 1 & \\ & 1 \end{bmatrix}$$

where $\mathbb{R}$ is a real number. It is clear that the four reflected symbols cannot fulfill the orthogonal property.

3.3 Why is VMscatter different?

In this section we describe our VMscatter's two unique features: i) it achieves the same full diversity gain as traditional MIMO system without transmitting orthogonal signals; and ii) our demodulation scheme estimates the complicated channels among sender, backscatter tag, and receiver while solving the demodulation errors caused by practical facts (e.g., cumulative clock drift and oscillator instability).

Feature 1: Although there is no orthogonal symbols for the MIMO backscatter to construct the space-time coding, we discover that MIMO backscatter can achieve the same full diversity gain as the space-time coding by using the non-orthogonal symbols.

Firstly, we define $\mathrm{M}(s_{ij}, s_{pq}) = \mathbf{H_T} \cdot \mathbf{diag}(s_{ij}^{-1}, s_{pq}^{-1}) \cdot \mathbf{H_T}^{-1}$. By multiplying $\mathrm{M}(s_{ij}, s_{pq})$ to both sides of Eqn. 4, we get:

$$\begin{aligned} \mathrm{M}(s_{1a}, s_{1b}) \begin{bmatrix} y_{1A} \\ y_{1B} \end{bmatrix} &= \mathbf{H_T} \cdot \begin{bmatrix} \psi_a \\ \psi_b \end{bmatrix} + \mathrm{M}(s_{1a}, s_{1b}) \begin{bmatrix} n_{1A} \\ n_{1B} \end{bmatrix} \\ \mathrm{M}(s_{2a}, s_{2b}) \begin{bmatrix} y_{2A} \\ y_{2B} \end{bmatrix} &= \mathbf{H_T} \cdot \begin{bmatrix} -\psi_b^* \\ \psi_a^* \end{bmatrix} + \mathrm{M}(s_{2a}, s_{2b}) \begin{bmatrix} n_{2A} \\ n_{2B} \end{bmatrix} \end{aligned} \quad (5)$$

We note that Eqn. 5 has the same standard representation as that of conventional 2×2 Alamouti MIMO systems (showed in Eqn. 2). In other words, if the value of $\mathrm{M}(s_{ij}, s_{pq})$ in Eqn. 5 is known, the MIMO backscatter system can also achieve full diversity gain, as proven in the following:

$$\begin{bmatrix} \psi_a & \psi_b \\ -\psi_b^* & \psi_a^* \end{bmatrix}^{\mathrm{H}} \cdot \begin{bmatrix} \psi_a & \psi_b \\ -\psi_b^* & \psi_a^* \end{bmatrix} = (|\psi_a|^2 + |\psi_b|^2) \cdot \begin{bmatrix} 1 & \\ & 1 \end{bmatrix} \quad (6)$$

To realize the space-time coding, the conjugate operations are required. Traditional RFID works by selecting different antenna impedance loading. To reduce the design complexity, VMscatter only leverages the two states of switch (on and off) to realize the conjugate operation (detailed in Sec. 4.2).

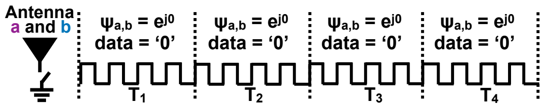
Feature 2: To solve Eqn. 5, we must know $\mathrm{M}(s_{ij}, s_{pq})$. However, to detect $\mathrm{M}(s_{ij}, s_{pq})$, we face two challenges. The first challenge is that the backscatter tag does not exactly shift the ambient signal as the desired phase in reality because of some practical facts (such as cumulative clock drift and oscillator instability). The difference of the desired shifting phase and the real shifting phase interfere the detection result of $\mathrm{M}(s_{ij}, s_{pq})$. The second challenge is that the existing channel estimation module cannot resolve both of the pre-scatter and post-scatter channels to extract the backscatter data. We will discuss how to overcome these two challenges in Sec. 4.3.

4 Design

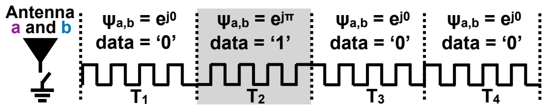
This section presents the detailed design of the VMscatter system. We first propose the channel model of the VMscatter system. Secondly, we introduce the modulation scheme at the VMscatter tag. Thirdly, we describe the demodulation scheme at the receiver. Finally, we explain how to extend from $2 \times 2 \times 2$ to $M \times K \times N$.

4.1 VMscatter Channel Model

In order to explain the signal propagations of our MIMO backscatter, we build a mathematical channel model for our system. Though a backscatter tag does not "receive" the signal (the tag "reflects" the signal out immediately when the signal comes in), we decompose the receiving and transmitting



(a) Example of transmitting data '0000'



(b) Example of transmitting data '0100'

Figure 3: Basic Modulation. The signal in the shaded time slot is shifted by π.

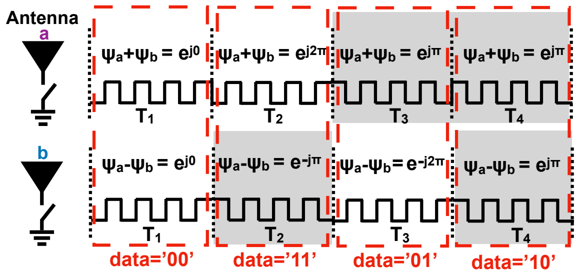


Figure 4: An example of improving throughput. Eight bits "00110110" are transmitted during four time slots. The signals in the shaded time slots are shifted by π.

processes in the channel model. Fig. 2(b) shows the channel model of a $2\times2\times2$ setup. Without loss of generality, let us take the receive vector in the first time slot (i.e., $[y_{1A}, y_{1B}]$) as an example to describe the channel model. From Eqn. 4 and 3, we can rewrite the representation of the receive vector in the first time slot as:

$$\begin{bmatrix} y_{1A} \\ y_{1B} \end{bmatrix} = \mathbf{H_T}\cdot\begin{bmatrix} s_{1a}\psi_a \\ s_{1b}\psi_b \end{bmatrix} + \begin{bmatrix} n_{1A}^{\mathrm{T}} \\ n_{1B}^{\mathrm{T}} \end{bmatrix} = \mathbf{H_T}\cdot\begin{bmatrix} \psi_a & \\ & \psi_b \end{bmatrix}\cdot\begin{bmatrix} s_{1a} \\ s_{1b} \end{bmatrix} + \begin{bmatrix} n_{1A}^{\mathrm{T}} \\ n_{1B}^{\mathrm{T}} \end{bmatrix} \quad (7)$$

Let us assume X, Y, Ψ, and N are the transmitted signal at the sender, the received signal at the receiver, the backscatter data, and the noise respectively. We rewrite Eqn. 7 here:

$$\mathrm{Y} = \mathbf{H_T}\Psi\mathbf{H_R}\mathrm{X} + \mathrm{N} \quad (8)$$

Eqn. 8 gives the channel model of a $2\times2\times2$ MIMO backscatter system. It proves that the backscatter modulation not only splits the whole channel (from sender to receiver) into two parts: $\mathbf{H_R}$ and $\mathbf{H_T}$, but also inserts the backscatter information into the whole channel. Therefore, to demodulate the backscatter data, it is important to estimate the channel $\mathbf{H_R}$ and $\mathbf{H_T}$. However, since existing channel estimators estimate the whole channel without considering the backscatter modulation, the estimation result is interfered by the backscatter data. To address this challenge, we first need to understand the backscatter modulation scheme.

4.2 MIMO Modulation @ VMscatter Tag

In this section, we introduce how a VMscatter tag creates MIMO signals which can increase the communication performance. Our goal is to realize MIMO transmission while maintaining low power consumption at a power constrainted VMscatter tag. The MIMO transmission can significantly reduce the bit error rate (BER) or increase the throughput by increasing the number of antennas. It is challenging to transmit spatial coded information at a lower power VMscatter tag to improve the communication performance at the receiver side. To understand it, we first introduce the basic modulation scheme with a $2\times2\times2$ setup, then describe two coding methods for improving the throughput and reducing the BER.

Basic modulation scheme: In a basic modulation scheme, the backscatter tag shifts the phase of an ambient OFDM signal on one antenna port or multiple antenna ports (transmitting same data on all ports) to transmit backscatter data. For example, the phase can be shifted by $e^{j\theta_k}$, where k is the index of antennas. $\theta_k = 0$ or π indicates data bit "0" or "1".

Fig. 3 shows a simple example that, in a $2\times2\times2$ setup, two antennas (a and b) transmit the same data by using RF switches (i.e., ADG902 [8]) to shift the ambient signal's phase. In Fig. 3(a), continuous square waves ($\theta_{a,b} = 0$) from T_1 to T_4 are driving the RF switches. Thus, no phase shift is caused to the ambient OFDM signal and the data of "0000" is transmitted. In Fig. 3(b), the square waves are shifted by $\theta_{a,b} = \pi$ during T_2 which yields the data of "0100" being transmitted.

Improving throughput: The VMscatter tag takes advantage of the MIMO system to increase throughput. For example, with two antennas, the throughput can be doubled. To increase throughput, the data is encoded across two antennas as follows: at time slot T_n, the aggregate value of two data streams (i.e., $\psi_a+\psi_b$) is transmitted from antenna a while the difference (i.e., $\psi_a-\psi_b$) is transmitted from antenna b simultaneously. Thus, the received signals at the receiver are:

$$\mathrm{Y} = \mathbf{H_T}\begin{bmatrix} \psi_a+\psi_b & \\ & \psi_a-\psi_b \end{bmatrix}\mathbf{H_R}\mathrm{X} + \mathrm{N} \quad (9)$$

Fig. 4 shows that 8 bits (i.e., "00110110") are encoded and transmitted on antenna a and b during time slots T_1 to T_4. The phase changes of $\psi_a+\psi_b$ and $\psi_a-\psi_b$ can be defined as $e^{j(a+b)}$ and $e^{j(a-2b)}$. For example, during T_2, data "11" is transmitted. Therefore, antenna a is transmitting $\psi_a+\psi_b = e^{j2\pi} = 1$ (i.e., zero phase change) while antenna b is transmitting $\psi_a-\psi_b = e^{-j\pi} = -1$ (i.e., a phase inversion).

Reducing BER: It is more challenging to reduce the BER on power constrained backscatter tag. In the VMscatter system, in order to have a more reliable transmission, we utilize the space-time coding on top of the basic modulation across all the antennas (i.e., the antennas at the VMscatter tag) and adjacent time slots. However, it is very challenging to implement space-time coding on the tag by only turning on/off the switches, because we do not want to increase the tag's computation and energy overheads. We note that since the backscatter data is real number, the conjugate of the backscatter data is itself. For example, $\psi_a = \psi_a^*$ and $-\psi_b^* = -\psi_b$. Therefore, VMscatter can construct the space-time coding by using the two states of switch (on and off). Fig. 5 shows the coding sequence. At a given time slot T_n, two streams of data are simultaneously transmitted from antennas a and b. The data transmitted from antennas a and b are denoted by ψ_a and ψ_b, respectively, where $\psi_a = e^{j\theta_a}$ and $\psi_b = e^{j\theta_b}$. During the

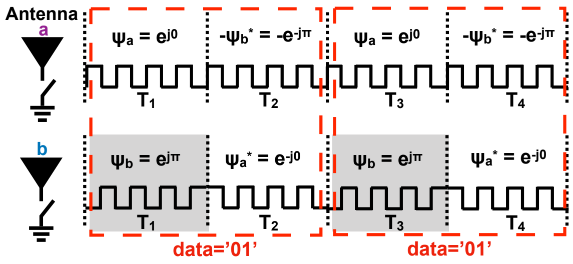


Figure 5: An Example of reducing BER. Four bits "0101" are transmitted during four time slots. The signals in the shaded time slots are shifted by π.

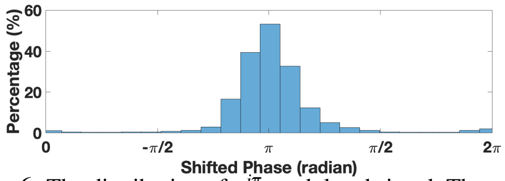


Figure 6: The distribution of $e^{j\pi}$ modulated signal. The sender-to-tag distance is 3 ft and the tag-to-receiver distance is 6 ft. We observe that the shifted phases are distributed between 0 and 2π.

next time slot T_{n+1}, data $-\psi_b^* = -e^{-j\theta_b}$ is transmitted from antenna a while data $\psi_a^* = e^{-j\theta_a}$ is transmitted from antenna b. By plugging the coding sequence into the channel model (Eqn. 8), the received signals in two time slots are:

$$\begin{aligned} Y_1 &= \mathbf{H_T}\begin{bmatrix}\psi_a & \\ & \psi_b\end{bmatrix}\mathbf{H_R}X_1 + N_1 \\ Y_2 &= \mathbf{H_T}\begin{bmatrix}-\psi_b^* & \\ & \psi_a^*\end{bmatrix}\mathbf{H_R}X_2 + N_2 \end{aligned} \tag{10}$$

Fig. 5 illustrates how four bits of data (i.e., "0101") are transmitted on antenna a and b during time slots T_1 to T_4. We take time slots T_1 and T_2 as an example. During T_1, the first data bit "0" (represented by $\psi_a = e^{j0}$) is transmitted from antenna a while the second data bit "1" (represented by $\psi_b = e^{j\pi}$) is transmitted from antenna b. During T_2, the encoded data (according to Eqn. 10) $-\psi_b^* = -e^{-j\pi}$ and $\psi_a^* = e^{-j0}$ are transmitted from antenna a and b, respectively.

4.3 MIMO Demodulation @ Receiver

In Sec. 4.2, we described the modulation scheme on a low power VMscatter tag. In this section, we will introduce how to demodulate the signal from a VMscatter tag. The most challenging part is to estimate the pre-scatter channel $\mathbf{H_R}$ (the channel between sender and backscatter) and post-scatter channel $\mathbf{H_T}$ (the channel between backscatter and receiver) in a MIMO setup, which is crucial for achieving reliable backscatter communication.

To demodulate the backscatter data, the **first challenge** is that in reality, the backscatter tag does not exactly shift the ambient OFDM signal by $e^{j\theta_k}$, which may yield demodulation error at the receiver. As we introduced in Sec. 4.2, the backscatter tag shifts the ambient signal by $e^{j\theta_k}$, where k is the index of the antennas, to modulate backscatter data. i.e. $\theta_k = 0$ indicates data bit "0", $\theta_k = \pi$ indicates data bit "1". However, due to some practical facts (i.e., cumulative clock

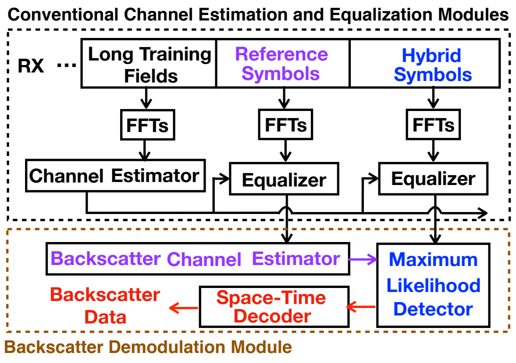

Conventional Channel Estimation and Equalization Modules
RX
Long Training Fields
Reference Symbols
Hybrid Symbols
FFTs
FFTs
FFTs
Channel Estimator
Equalizer
Equalizer
Backscatter Channel Estimator
Maximum Likelihood Detector
Backscatter Data
Space-Time Decoder
Backscatter Demodulation Module

Figure 7: Proposed backscatter demodulation module. The backscatter demodulation module takes the output of the conventional channel estimation and equalization modules as input.

drift and oscillator instability), the backscattered signal is not exactly shifted by $e^{j\theta_k}$. Fig. 6 shows the distribution of a backscattered signal sample set. The desired phase shift is $e^{j\theta_k}$, where $\theta_k = \pi$. The total symbols of the sample set are more than 800,000. We can observe that around 50% of the symbols are shifted by π while another 50% are shifted either less or more than π.

The **second challenge** is that the existing channel estimation method (i.e., simply solving the channel matrix) cannot resolve both of the pre-scatter channel matrix $\mathbf{H_T}$ and post-scatter channel matrix $\mathbf{H_R}$, which is important to demodulate the backscatter data (as we modeled in Eqn. 8). As shown in Fig. 7, the conventional channel estimation module relies on the Long Training Fields (LTFs) in the preamble to estimate the channel between the sender and receiver. Then the equalizer uses the estimated channel matrix to compensate for the upcoming data symbol. However, with a backscatter tag in the middle (of the sender and the receiver), we need to resolve both channel matrices $\mathbf{H_T}$ and $\mathbf{H_R}$ (as shown in Eqn. 11) for reliable demodulation.

To overcome the **first challenge**, we have the following analysis. Though the two practical facts (cumulative clock drift and oscillator instability) cause the frequency mismatch between the VMscatter tag and OFDM receiver, these two facts behave as a phase offset in the demodulation procedure. Fig. 6 shows the distribution of a sample set of the backscatter signal. The expected phase is π because the VMscatter tag transmits a $e^{j\pi}$. However, due to the cumulative clock drift and oscillator instability, a portion of the symbols are not as expected. To show the impact of these two practical facts, we elaborate Eqn. 8 to be:

$$Y = \mathbf{H_R}\Psi P\mathbf{H_T}X + N = \mathbf{H_R}\Psi\mathbf{H'_T}X + N \tag{11}$$

where P is the unwanted phase offset caused by cumulative clock drift and oscillator instability. Since the essences of P and $\mathbf{H_T}$ are all phase offsets, we can combine them as $\mathbf{H'_T} = P\mathbf{H_T}$. So far, we have solved the **first challenge** and the **second challenge** becomes resolving $\mathbf{H_R}$ and $\mathbf{H'_T}$.

To overcome this challenge, we propose a special backscatter packet structure (shown in Fig. 8) as well as a backscatter

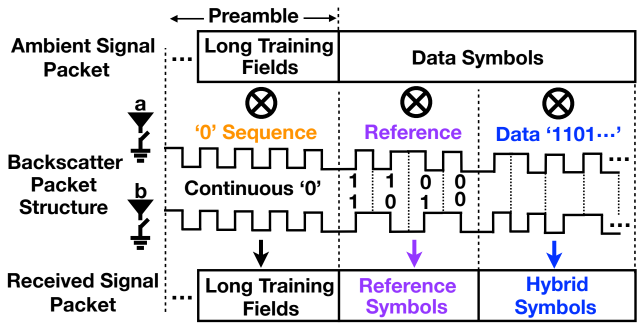


Figure 8: VMscatter packet structure. It includes three fields: i) 0s sequence, in which the VMscatter transmits continuous "0" on both antennas to pass through the LTFs from the ambient signal; ii) reference signal, in which the VMscatter transmitted agreed symbols on two antennas; and iii) backscatter data.

demodulation module on top of the conventional channel estimation and equalization modules (shown in Fig. 7). The packet includes three fields: **i) "0" sequence**; **ii) reference signal**; and **iii) backscatter data**.

i) "0" sequence: The VMscatter tag transmits a "0" sequence during the ambient signal's Long Training Fields (LTFs). The "0" sequence can shift the LTFs to the adjacent channel for minimizing the strong self-interference from the original channel and getting a high SNR [63]. However, it does not embed any useful backscatter data in the LTFs. Thus, the backscatter data Ψ in this filed equals to an identity matrix I_2 and Eqn. 11 becomes:

$$Y_{LTFs} = \mathbf{H_R} I_2 \mathbf{H'_T} X_{LTFs} + N = \mathbf{H_E} X_{LTFs} + N \tag{12}$$

where $\mathbf{H_E} = \mathbf{H_R} I_2 \mathbf{H'_T} = \mathbf{H_R H'_T}$ is the channel response for the transmitted LTFs (i.e., X_{LTFs}). Since X_{LTFs} are known to the receiver, $\mathbf{H_E}$ is resolved by the existing channel estimator (shown in Fig. 7). Then it is used to equalize the data symbol in existing equalizers by using the following equation:

$$\widetilde{X} = \mathbf{H_E^{-1}}\, Y = \mathbf{H_T'^{-1}}\, \mathbf{H_R^{-1}}\, Y \tag{13}$$

where $\widetilde{X}$ is the estimated symbol value. By plugging Eqn. 11 into Eqn. 13, we will get:

$$\widetilde{X} = \mathbf{H_T'^{-1}} \cancel{\mathbf{H_R^{-1}}} \cancel{\mathbf{H_R}} \Psi \mathbf{H'_T} X + \mathbf{H_E^{-1}} N = \mathbf{H_T'^{-1}} \Psi \mathbf{H'_T}\, \mathbf{X} + \mathbf{H_E^{-1} N} \tag{14}$$

We note that the impact of the pre-scatter channel $\mathbf{H_R}$ is eliminated so far by using the "0" sequence and the existing channel estimation and equalization process. To obtain Ψ, we need to resolve $\mathbf{H'_T}$. $\mathbf{H_E^{-1}}$N is random noise which can be resolved by a Maximum Likelihood estimation.

ii) Reference signal: To resolve $\mathbf{H'_T}$, we designed the second field **Reference signal** in the VMscatter packet structure (Fig. 8). A group of agreed on (between VMscatter tag and OFDM receiver) symbols are transmitted to estimate $\mathbf{H'_T}$. The backscatter channel estimation module (in Fig. 7) calculates the estimated $\widetilde{\mathbf{H'}}_\mathbf{T}$ by using:

$$\widetilde{\mathbf{H'}}_\mathbf{T} = \underset{\mathbf{H'_T}}{\arg\min} \sum_{\Psi_{ref}} ||\widetilde{X} - \mathbf{H_T'^{-1}}\, \Psi_{ref}\, \mathbf{H'_T}\, X||^2 \tag{15}$$

where Ψ_{ref} is the backscatter reference symbols as shown in Fig. 8. X can be obtained from the original TX-RX channel

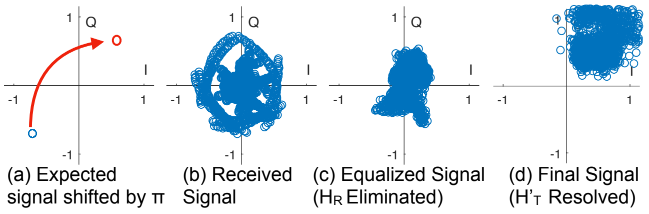


Figure 9: The Example Signals at Each Demodulation Stage.

without backscatter's interference (similar techniques are proposed in [61, 62]). Eqn. 15 is a minimum mean squared error (MMSE) channel estimator. A reduced complexity MMSE estimator can be achieved by employing the optimal rank reduction [60], which has a computational complexity of $O(LSC^2)$. Where L is the number of reference symbols, S is the number of subcarriers and $C = min(K,N)$. For a low power edge computing platform Jetson Nano [45] (only cost $99) implemented on an ARM A57 processor with 472 Giga -Floating Point Operations per Second, it can solve a 4×4 MIMO channel estimation (with 4 reference symbols and 64 subcarriers) in $\frac{4*64*4^2}{472*10^9} = 8ns$, which is shorter than a symbol duration $4\mu s$.

iii) Data: After resolving $\mathbf{H'_T}$, a maximum likelihood estimation module (shown in Fig. 7) is used to calculate the upcoming data field. The maximum likelihood estimation module minimizes the Euclidean distance between the equalized values $\widetilde{X} = \begin{bmatrix} \tilde{x}_A & \tilde{x}_B \end{bmatrix}^\intercal$ and all possible estimated values $\widetilde{X}_{MLE} = \begin{bmatrix} \tilde{x}_{A\_MLE} & \tilde{x}_{B\_MLE} \end{bmatrix}^\intercal = \widetilde{\mathbf{H'}}_\mathbf{T}^{-1}\, \Psi\, \widetilde{\mathbf{H'}}_\mathbf{T}\, X$, where $\tilde{x}_A$ and $\tilde{x}_B$ are the values on RX antenna A and B, respectively. $\widetilde{X}$ is obtained from Eqn. 14. $\widetilde{\mathbf{H'}}_\mathbf{T}$ is obtained from Eqn. 15. $\Psi = \text{Diag}(e^{j\theta_a}, e^{j\theta_b})$ is the possible backscattered data, X is obtained from the original TX-RX channel as we mentioned before. Thus, the backscatter data can be obtained by the following equation:

$$\begin{bmatrix} \tilde{\theta}_a \\ \tilde{\theta}_b \end{bmatrix} = \underset{\theta_a,\theta_b \in \{0,\pi\}}{\arg\min} \sum_{s=A,B} \sum_{t=0}^{T} ||\tilde{x}_s(n+t),\, \tilde{x}_{s\_MLE}(n+t)||^2 \tag{16}$$

where $T = 1$ for VMscatter low bit error rate mode and $T = 0$ for VMscatter high throughput mode.

After demodulating $[\tilde{\theta}_a, \tilde{\theta}_b]^\intercal$, the space-time decoding module calculates the data bit value based on the coding method as proposed in Sec. 4.2.

Fig. 9 shows example signals at different demodulation stages. In Fig. 9(a), the blue circle indicates the I/Q values of the ambient signal while the red circle indicates that the signal is shifted by π after backscattering. Fig. 9(b) shows the received signal without any equalization. We can observe the rotating pattern which may be caused by channel and frequency mismatch. Fig. 9(c) shows the signal passed through the conventional channel estimation and equalization module, in which the pre-scatter channel $\mathbf{H_R}$ is eliminated. Finally, the signal in Fig. 9(d) is processed by our backscatter demodulation module where $\mathbf{H'_T}$ (which includes the post-scatter channel $\mathbf{H_T}$ and cumulative clock drift and oscillator instability P) is resolved. We can observe that most dots are in the first quadrant, which can be correctly demodulated.

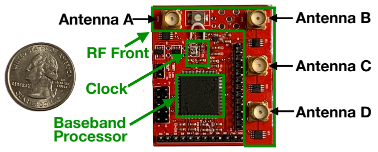


Figure 10: The hardware tag of VMscatter.

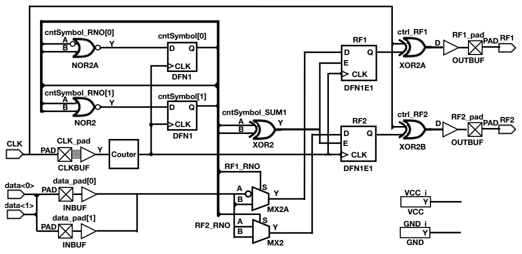


Figure 11: VMscatter tag hardware diagram.

4.4 From $2 \times 2 \times 2$ to $M \times K \times N$

In this section, we extend the backscatter MIMO system from $2 \times 2 \times 2$ to $M \times K \times N$.

Modulation. To reduce BER, in Sec. 4.2, we already show the 2×2 coding matrix (i.e., Matrix 17 shown below).

$$\Gamma(\theta_a, \theta_b) = \begin{bmatrix} e^{j\theta_a} & e^{j\theta_b} \\ -e^{-j\theta_b} & e^{-j\theta_a} \end{bmatrix} \quad (17)$$

When there are 4 antennas on the backscatter ($K = 4$), the transmitted backscatter data becomes $\{\theta_a, \theta_b, \theta_c, \theta_d\}$. Then, from Matrix 17, the coding matrix will be extended as follows:

$$\Gamma(\theta_a, \theta_b, \theta_c, \theta_d) = \begin{bmatrix} \Gamma(\theta_a, \theta_b) & \Gamma(\theta_c, \theta_d) \\ -\Gamma^*(\theta_c, \theta_d) & \Gamma^*(\theta_a, \theta_b) \end{bmatrix} \quad (18)$$

Each element in the coding Matrix 18 can be calculated from Matrix 17. In other words, for the backscatter with 4 antennas, each element in the coding matrix can be represented by the coding matrix of the backscatter with 2 antennas. Therefore, for the backscatter with K antennas, the coding matrix can be represented as follow:

$$\Gamma(\theta_1, \ldots, \theta_K) = \begin{bmatrix} \Gamma(\theta_1, \ldots, \theta_{\frac{K}{2}}) & \Gamma(\theta_{1+\frac{K}{2}}, \ldots, \theta_K) \\ -\Gamma^*(\theta_{1+\frac{K}{2}}, \ldots, \theta_K) & \Gamma^*(\theta_1, \ldots, \theta_{\frac{K}{2}}) \end{bmatrix} \quad (19)$$

In this coding matrix, each element can be calculated from the coding matrix of backscatter with $K/2$ antennas.

Demodulation. In Sec. 4.3 Eqn. 15, we have built the backscatter channel estimation model to estimate the prescatter channel $\mathbf{H}'_\mathbf{T}$. In this model, $\mathbf{H}'_\mathbf{T}$ is a 2×2 matrix and has the invertible matrix $\mathbf{H}'^{-1}_\mathbf{T}$. For a $M \times K \times N$ setup, $\mathbf{H}'_\mathbf{T}$ can be represented as a $M \times K$ matrix. When $M = K$, $\mathbf{H}'_\mathbf{T}$ is invertible and $\mathbf{H}'^{-1}_\mathbf{T}$ exists. When $M \neq K$, the invertible matrix can be replaced by the pseudoinverse matrix $(\mathbf{H}'^{\mathrm{T}}_\mathbf{T}\mathbf{H}'_\mathbf{T})^{-1}\mathbf{H}'^{\mathrm{T}}_\mathbf{T}$ [20]. Therefore, we can still leverage the backscatter channel estimation model and the maximum likelihood estimation model to estimate $\mathbf{H}'_\mathbf{T}$ and further demodulate the backscatter data.

5 Evaluation

5.1 Implementation

Hardware Tag. We implement VMscatter on a customized four-layer printed circuit board (PCB). As shown in Fig. 10, the hardware tag has three components: a $20MHz$ clock oscillator, a baseband processor and an RF front. In the evaluation, we show the impact of different oscillator accuracies.

We use a low-power FPGA (Microsemi Igloo Nano AGLN250 [40]) as the baseband processor. The processor controls the RF front, which consists of 4 RF switches (ADG902) and connects up to 4 antennas. All the design introduced in Sec. 4.2 is implemented in the processor. Fig. 11 shows a basic VMscatter that can support 2 antennas.

Comparing with conventional backscatter systems [62, 68], VMscatter only needs to add several low-power RF switches and antennas to build the MIMO system. These switches and antennas can be easily implemented on conventional backscatter systems. The space-time coding can be achieved by shifting the phase of the backscatter data to its orthogonal alternative, which also can be easily employed on the conventional backscatter system. To be more specific, as mentioned in Passive WiFi [34], the phase shift can be realized by shifting the initial phase of the square wave on the switch. As a result, the modulation complexity of VMscatter is low.

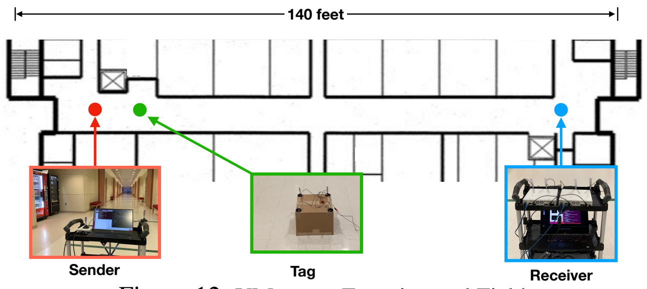


Figure 12: VMscatter Experimental Field

Sender and Receiver Implementation. As shown in Fig. 12, we implement the MIMO-OFDM sender on a B210 USRP, which connects two antennas and is placed 3 feet away from the tag. The sender's output power is set as 0 dBm. The receiver is implemented on two X310 USRPs with four $9dbi$ omnidirectional antennas. Therefore, there is a 2×4 MIMO across the sender and receiver. Each receiving antenna is connected to a UBX-160 daughterboard, which down-converts the backscattered signal to the baseband signal and samples it. To synchronize and align the sample clocks, the two USRPs are connected in a daisy chain configuration [1], where one device in the chain is configured as a master and exports its 10MHz clock and PPS time references to the other device.

5.2 Experiment Setup

To extensively evaluate the performance of VMscatter, we conducted the experiments in the following configurations:

Single Input Multiple Output (SIMO): The sender has a single data stream (therefore a single transmitting antenna). We evaluate the performance of VMscatter in SIMO with different number of backscatter antennas (2 and 4), which constructs a $1 \times 2 \times 2$ and a $1 \times 4 \times 4$ setup, respectively.

Multiple Input Multiple Output (MIMO): The sender has two independent data streams (therefore two transmitting antennas). We evaluate the performance of VMscatter in MIMO

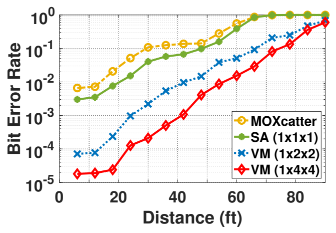


Figure 13: BER

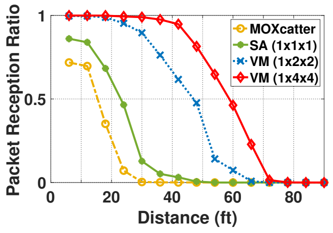


Figure 14: PRR vs. Distance

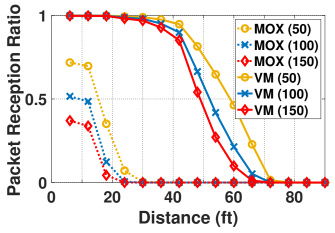


Figure 15: PRR vs. Packet Length

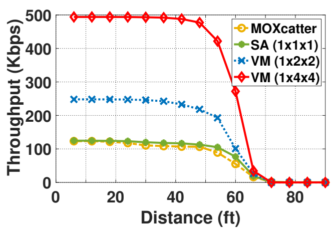


Figure 16: Throughput

with different number of antennas (2 and 4), which constructs a $2 \times 2 \times 2$ and a $2 \times 4 \times 4$ setup, respectively.

Baseline. We implemented the state-of-the-art MIMO backscatter system (MOXcatter [68]) as our baseline. Because it neither employs the space-time coding nor introduces any backscatter channel estimation. We also implemented a $1 \times 1 \times 1$ Single Antenna setup (SA), which only employs the backscatter channel estimation technique. Comparing it with MOXcatter, the performance of the backscatter channel estimation is evaluated. Comparing it with VMscatter, the performance of space-time coding is evaluated.

5.3 VMscatter in SIMO

In this section, we show the evaluation results of VMscatter with one transmitting antenna (i.e., single data stream).

5.3.1 Bit Error Rate

Fig. 13 shows the BER of VMscatter under the low bit error rate mode. The performance of VMscatter is much better than that of MOXcatter. When the distance between backscatter and the receiver is as low as $6ft$, the BER for VMscatter with 4 antennas is around 0.00002, which is reduced by a factor of 366 compared to MOXcatter (0.007). As the distance increases to $66ft$, the BER of VMscatter is still low enough to conduct communication while the BER of MOXcatter is 1. The reasons why VMscatter shows dominant performance is because VMscatter uses a novel backscatter channel estimation technique and leverages the spatial diversity for communication, which eliminates the errors introduced by cumulative clock drift or oscillator instability, etc and significantly reduce the BER. In addition, we can also observe that by simply using channel estimation, the BER of a $1 \times 1 \times 1$ single antenna setup (SA) is better than that of MOXcatter with multiple antennas. In contrast, although MOXcatter can backscatter a single stream from the WiFi sender, it does not leverage the spatial diversity in the reflected signals and channel estimation techniques, which increases the BER.

5.3.2 Packet Reception Ratio

Fig. 14 shows the Packet Reception Ratio (PRR) of VMscatter (low bit error rate mode) under different distances between backscatter and receiver. The PRR of MOXcatter drops rapidly as the distance increases from $6ft$ to $30ft$ while the performance of SA is still better than that of MOXcatter. For the distances larger than $30ft$, nearly all the packets transmitted from MOXcatter cannot be correctly decoded at the receiver side. On the contrary, since SA utilizes channel estimation, it can still conduct communication until the distance reaches $50ft$. Our VMscatter shows great advantages as distance increases. The curves are much more stable for both the 2 antennas and 4 antennas configurations. For the first $30ft$, the PRR of VMscatter is more than 90%, which is much higher than that of MOXcatter.

As shown in Figure 15, we also study the Packet Reception Ratio (PRR) of VMscatter (low bit error rate mode) with the packet length varying from 50 bits to 150 bits. Since the experiments reveal similar trends, we only show the results of the 4 antennas configuration (1x4x4). We can observe that the PRR decreases with the increasing of the packets length while VMscatter is more robust against different packet lengths. As shown in this figure, the PRR reaches nearly 0 as the distance increases to $70ft$. For MOXcatter, it does not use any space-time coding and channel estimation techniques. As a result, the PRR is nearly 0 when the distance increases to $30ft$ for the 50 bits packet length. However, when the packet lengths are 100 and 150 bits, the PRR is nearly 0 as the distance increases to $24ft$. Therefore, VMscatter can effectively increase the communication range of backscatter.

5.3.3 Throughput

We show the throughput of VMscatter under the high throughput mode with the increasing of the distance between backscatter and receiver. As we can see from Fig. 16, the throughput of VMscatter with two antennas is around 2 times as high as that of MOXcatter. This is because VMscatter takes advantages of the MIMO communication techniques. Specifically, the antennas on VMscatter are transmitting different data, which doubles the throughput. As the number of transmitting antennas increases to 4, the throughput of VMscatter is around 4 times as high as that of MOXcatter. On the contrary, the antennas on MOXcatter are transmitting the same data, which wastes the communication resources. In addition, due to the lack of channel estimation, the throughput is further hampered. As a result, the performance of MOXcatter is even worse than that of single antenna setup (SA).

5.4 VMscatter in MIMO

In this section, we evaluate the performance of VMscatter with two transmitting antennas (i.e., multi-data stream).

5.4.1 Bit Error Rate

As shown in Fig. 17, similar to a single data stream, the BER of VMscatter is still much lower than that of MOXcatter and SA. When the distance is as low as $6ft$, the BER of VMscatter is around 0.000011, which is reduced by a factor of 862 compared to the BER of MOXcatter. When comparing with

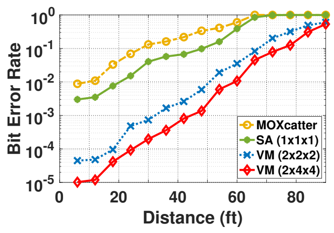


Figure 17: BER

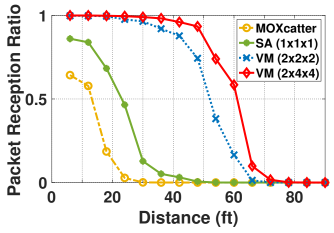


Figure 18: PRR vs. Distance

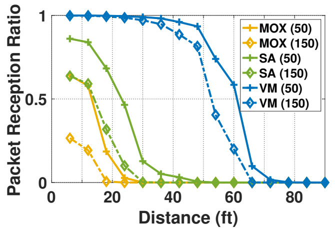


Figure 19: PRR vs. Packet Length

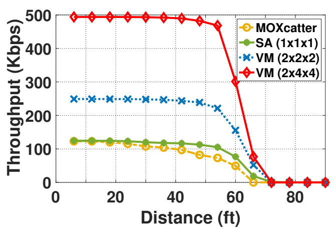


Figure 20: Throughput

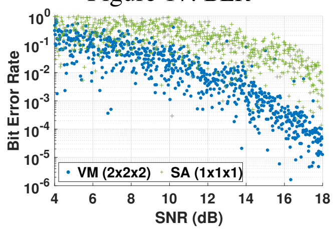


Figure 21: BER v.s. SNR

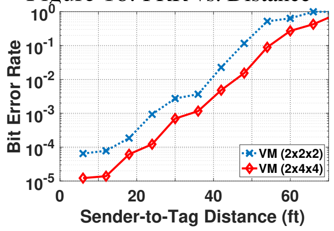


Figure 22: BER with a fixed tag-to-receiver distance

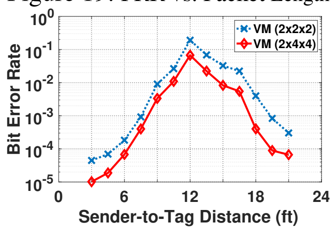


Figure 23: BER with a fixed sender-to-receiver distance

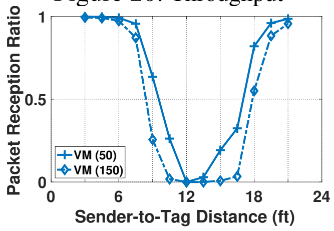


Figure 24: PRR with a fixed sender-to-receiver distance

the single data stream scenario in Fig. 13, we can also observe that the BER of VMscatter with two transmitting antennas is lower than the BER with one transmitting antenna. Interestingly, as shown in Table 1, the BER of MOXcatter shows the opposite trends, that is, the BER with two transmitting antennas is higher than the BER in the single data stream scenario and even higher than BER of the non-MIMO WiFi backscatter system FreeRider [62].

Table 1: BER Comparison

System	FreeRider (Non-MIMO)	MOXcatter (MIMO)	VMscatter (MIMO)
BER	0.002	0.0095	0.000011

This is because as the number of transmitting antennas increases, the interference and multi-path effects become more severe than the single antenna scenario, which degrades the performance of MOXcatter. In contrast, VMscatter fully leverages the feature of the MIMO spatial diversity. As the number of transmitting antennas increases, it is easier for VMscatter to conduct communication. Therefore, different from MOXcatter, VMscatter shows great advantages in this scenario.

5.4.2 Packet Reception Ratio

Fig. 18 and Fig. 19 show the Packet Reception Ratio under various distances and packet lengths, respectively. Similar to the single antenna scenario, the packet reception ratio of VMscatter is much higher than that of MOXcatter and SA. We can observe that the performance of SA with 150 bits packet length is better than that of MOXcatter with 50 bits packet length. This results prove the advances of our channel estimation technique. Moreover, we also can observe that the packet reception ratio of VMscatter is almost 100% for the first $36ft$, which shows the reliability of the low bit error rate mode of VMscatter.

5.4.3 Throughput

Fig. 20 shows the throughput of VMscatter with 2 and 4 backscatter antennas under the high throughput mode. The throughput of MOXcatter is even worse than that of the single antenna SA. This is because MOXcatter does not introduce any channel estimation techniques. As the number of transmitting antennas increases, the interference is higher, which reduces the throughput. In contrast, VMscatter can estimate the channel to minimize the interference and other unwanted errors. Moreover, VMscatter utilizes the MIMO technique to conduct backscatter communication. As the number of backscatter antennas increases to 2 and 4, the throughput of VMscatter also increases, which is around 2 and 4 times as high as that of MOXcatter.

5.4.4 BER v.s. SNR

To better understand the gain of MIMO technique for backscatter communication, we investigated the relationship between BER and SNR. In this experiment, we fixed the locations of sender, backscatter tag, and receiver while varying the transmission power of the excitation signals. Figure 21 plots the captured data points of BER over SNR for VMscatter in a $2\times2\times2$ (VM) configuration comparing with the signal antenna configuration (SA). We can observe that the BER is around 3×10^{-5} which is at least two orders of magnitude lower than that of SA.

5.5 Impact

In this section, we evaluate VMscatter under a variety of settings to further show the advantages of our design.

5.5.1 Impact of Sender-to-Tag Distance

The sender-to-tag distance is an important factor for the backscatter communication range. We evaluate its impact in two sets of experiments. In the first experiment, we fix the tag-to-receiver distance to be $1ft$ and move the tag and receiver to different locations, searching for the maximum backscatter communication range. As shown in Fig. 22, when the sender-to-tag distance increases to $45ft$, the BER for VMscatter with 4 antennas is around 0.01, which is still low enough to conduct communication.

In the second experiment, we fix the location of the sender and place the receiver at a distance of 24 ft. Then, we move the tag along the line from the sender to the receiver and measure

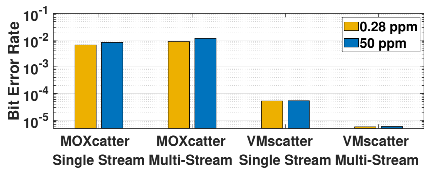


Figure 25: Impact of Oscillator Accuracy

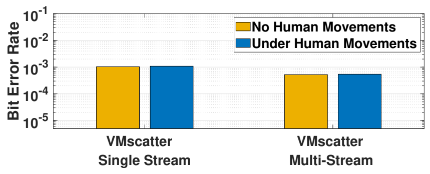


Figure 26: BER vs. Human Movement

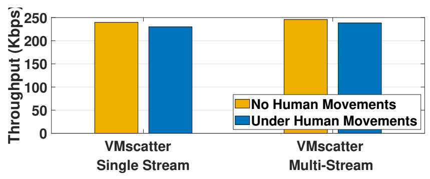


Figure 27: Throughput vs. Human Movement

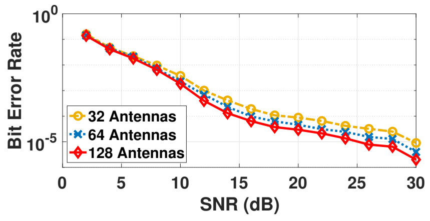


Figure 28: Number of Sender's Antennas

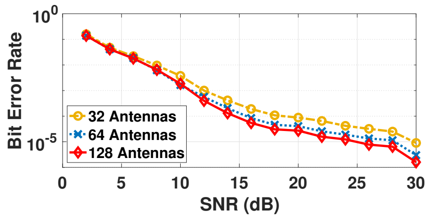


Figure 29: Number of Receiver's Antennas

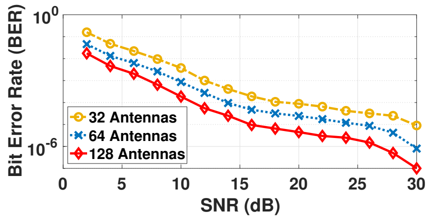


Figure 30: Number of Backscatter's Antennas

the BER and the packet reception ratio reported by VMscatter. Fig. 23 and Fig. 24 show the BER and packet reception ratio results, which can be improved either by reducing the sender-to-tag distance or the tag-to-receiver distance. This is because the signal strength at the receiver can be modeled by using Friis Transmission Equation [34, 42]. As the sender-to-tag distance or the tag-to-receiver distance decreases, the signal strength is increased at the receiver side, which means the SNR is also improved.

5.5.2 Impact of Backscatter Oscillator Accuracy

We study the impact of oscillator accuracy under VMscatter low bit error rate mode. The distance between the backscatter and the receiver is $6ft$. As shown in Fig. 25, with the decrease of crystal oscillator accuracy, the BER of MOXcatter increases while the BER of VMscatter almost remains the same. This is because MOXcatter is suffering cumulative clock drift and oscillator instability. Therefore, the performance of MOXcatter is highly related to the oscillator accuracy. In practice, an oscillator with high accuracy is more expensive and power hungry, which will limit the potential low power applications of MOXcatter. In contrast, our VMscatter utilizes a channel estimation technique, which eliminates the errors introduced by oscillator. In summary, VMscatter is stable against oscillator accuracy. A low power ring oscillator is enough for VMscatter to achieve low BER. Therefore, VMscatter has the potential to be widely adopted to real world scenarios.

5.5.3 Impact of Human Movements

Fig. 26 (low bit error rate mode) and Fig. 27 (high throughput mode) study the performances of VMscatter with human movements. During the experiments, the distance between the backscatter and the receiver is $36ft$. Two people are walking between the backscatter and the receiver. As shown in Fig. 26, the performance of VMscatter is robust against human movements. The BER remains at the level of 10^{-3} for single data stream and 10^{-4} for multi-data stream, respectively. Fig. 27 shows the throughput of VMscatter under high throughput mode. As we can see from this figure, when there are no human movements, the throughputs of VMscatter are $239.7Kbps$ and $245.9Kbps$ for single data stream and multi-data stream, respectively. For the human movements scenario, the throughputs of VMscatter still remain at $230.2Kbps$ and $238.5Kbps$, which further demonstrates the reliable communication of our VMscatter.

5.6 Simulation

To demonstrate that VMscatter can work under massive MIMO scenarios, we show the simulation results in this section. In the simulation, the physical layer configurations of the sender and receiver follow the 802.11n specification [2].

Fig. 28 and Fig. 29 study the BER with the increase of signal to noise ratio (SNR) under different number of antennas configurations. These two figures show a similar trend, that is, the BER decreases as the SNR increases. Moreover, the BER of VMscatter is lower when the number of antennas increases. When SNR reaches $30dB$, the BER results are significantly reduced (at the level of 10^{-5}). These results show that VMscatter can achieve better performance in massive MIMO scenarios, especially when the numbers of sender and receiver antennas are large.

Fig. 30 shows the results with different number of antennas on a backscatter device. When the number of antennas increases, the BER decreases. When the SNR is as low as $2dB$, the BER for 32 antennas is at the level of 10^{-1} while the BERs for 128 antennas are at the level of 10^{-2}. As SNR increases to $30dB$, the BER for 32 antennas is at the level of 10^{-5}. In contrast, the BER for 64 antennas drops to the level of 10^{-6} while the BER for 128 antennas drops below the level of 10^{-6}. These results further demonstrate that VMscatter can effectively leverage the spatial diversity to conduct reliable backscatter communication.

5.7 Energy Consumption

In this section, we first measure the power consumption of our hardware tag. We then design an integrated circuit based on the hardware and conduct power simulations to show the potential low-power capability.

5.7.1 Hardware Power Consumption Measurement

The hardware power consumption of VMscatter tag is similar to those traditional backscatter systems [61, 62, 69] that do not support MIMO communication. First, VMscatter only has $1 \sim 3$ more *passive* antennas and low-power RF switches than traditional backscatter systems. Therefore, the power consumption on the RF front is low. Second, the low-power FPGA [40] provides an internal ring oscillator to generate the clock, which consumes significantly less power than an external oscillator [64]. Third, the low-power FPGA supports the flash freeze technology. According to our lightweight modulation technique, 80% flash can be frozen.

We utilize a KEITHLEY 2701 multimeter to measure VMscatter tag's DC power draw. Under the work mode, the tag consumes as low as $464\mu W$ (it is $32\mu W$ in IC design simulation), which is $516\times$ lower than existing 802.11n MIMO chip [51]. Under the sleep mode, it consumes as little as $2.4\mu W$ of power, which is $25\times$ lower than existing 802.11n MIMO chip [51]. We can find that the work mode saves much more power than that of the sleep mode when comparing with MIMO chip. This is because the advantage of the backscatter is passive radio communication while the WiFi chip needs to generate the active radio under work mode. If the duty-cycles of the backscatter are from 1% to 10%, the overall power consumptions are around $5\mu W$ to $50\mu W$.

5.7.2 IC Design Power Consumption Simulation

To show the potential low-power capability of VMscatter, we designed an integrated circuit (IC) for VMscatter tag. The IC design consists of four main components: RF transistors, modulation logic gates, coding logic gates, and a ring oscillator. The simulation is conducted by using the HSPICE model for TMSC 55nm process. In the power consumption simulation, we considered multiple factors including operating voltages, operating temperature, system clock frequency, and power mode usage. Overall , the results show that the power consumption of VMscatter tag IC under work mode is $32\mu W$.

6 Related Work

The related work can be divided into two categories:

Backscatter Communications. Backscatter is one of the hot topics in recent years for its potential to support low-power and low-cost applications [5, 24, 38, 41, 52, 64]. Lots of work has been proposed to support various types of backscatter communications, such as WiFi [33, 59], Bluetooth [30], TV [39], FM [53]. LoRa [28, 47], or even Quantum Backscatter Communication [22] etc. Furthermore, researchers also improved backscatter to support full duplex [13]. The most related work to VMscatter is WiFi backscatter techniques. The first work of backscattering WiFi signals is WiFi Backscatter [33], which mainly modulates the CSI/RSSI information to conduct backscatter communication. Based on this work, BackFi [14] modulates phase information of the received WiFi signal and leverages the full-duplex technique to improve the throughput and communication range at the same time. HitchHike [61] introduces a codeword translation technique to make backscatter communication compatible with 802.11b radios. To support OFDM transmissions, [58] introduces a system model that analyzes the OFDM backscatter. FreeRider [62] can support backscatter communications with OFDM signals by leveraging the codeword translation to OFDM signals. MOXcatter [68] is able to backscatter MIMO signals from the WiFi MIMO sender. However, the multiple antennas on MOXcatter do not take advantage of the advanced features of MIMO technology.

MIMO Techniques. MIMO techniques have become one of the most important part in modern communication systems. Lots of projects have been proposed with focus on full duplex MIMO [10, 11, 15], Multi-User MIMO [9, 19, 35, 48–50, 56], Massive MIMO [12, 29, 32, 66], MIMO networks [21, 25, 57, 67] and novel MIMO systems [16–18, 37, 46], etc. For example, to improve the energy efficiency, CMES [46] mainly targets at finding the efficient antenna settings for MIMO 802.11 devices. To study full duplex MIMO, [15] introduces a full duplex WiFi-PHY based MIMO radios while [11] compares the capacity of multiple-antennas half-duplex MIMO with a full-duplex MIMO. Researchers have also presented lots of systems with focus on Multi-User MIMO (MU-MIMO) and Massive MIMO. MUSE [50] proposes a user selection framework on commodity devices to improve the throughput gains while Hekaton [56] combines a beamforming technique with phased-array antennas to improve the capacity gains in large-scale MU-MIMO systems. [66] introduces a scalable directional training method to obtain CSI in FDD massive MIMO systems. Surface MIMO [16] leverages the conductive paint or cloth on the surface to create an additional spatial path for communication between small devices.

Our work builds on top of the existing techniques in backscatter and MIMO. To the best of our knowledge, VMscatter is the first work that designed and implemented a versatile MIMO system. By doing this, VMscatter effectively leverages the spatial diversity feature of ambient MIMO signals, which significantly reduces the bit error rate and improves the throughput. Moreover, our design is generic and has the potential to be extended to massive MIMO.

7 Conclusion

In this paper, we present a new MIMO system on the backscatter device that can i) support various number of antennas at the sender, backscatter, and receiver; ii) significantly reduce the bit error rate or increase the throughput; and iii) achieve a similar level of energy consumption as existing backscatter systems. We built a hardware prototype and extensively evaluated our system under various real-world settings to demonstrate its extremely low BER performance. For example, the BER is reduced by a factor of 862 compared to the most related work MOXcatter [68].

Acknowledgments

This project is supported by NSF grants CNS-1652669 and CNS-1824491. We also thank anonymous reviewers and our shepherd Dr. Dinesh Bharadia for their valuable comments.

References

[1] Ettus research usrp hardware driver and usrp manual. https://files.ettus.com/manual/page_multiple.html. [Online].

[2] Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, 2016.

[3] Microstrip antenna — Wikipedia. https://en.wikipedia.org/wiki/Microstrip_antenna, 2019. [Online].

[4] Radio-frequency microelectromechanical system — Wikipedia. https://en.wikipedia.org/wiki/Radio-frequency_microelectromechanical_system, 2019. [Online].

[5] Ali Abedi, Mohammad Hossein Mazaheri, Omid Abari, and Tim Brecht. Witag: Rethinking backscatter communication for wifi networks. In *HotNets*, 2018.

[6] C.J. Aguilar-Armenta and S.J. Porter. Cantilever rf-mems for monolithic integration with phased array antennas on a pcb. *International Journal of Electronics*, 2015.

[7] S. M. Alamouti. A simple transmit diversity technique for wireless communications. *IEEE Journal on Selected Areas in Communications*, 1998.

[8] Analog Devices. *ADG901/ADG902: Wideband, 40 dB Isolation at 1 GHz, CMOS 1.65 V to 2.75 V, SPST Switches Data Sheet*, 2019. Rev. D.

[9] Narendra Anand, Jeongkeun Lee, Sung-Ju Lee, and Edward W Knightly. Mode and user selection for multi-user mimo wlans without csi. In *INFOCOM*, 2015.

[10] Ehsan Aryafar, Mohammad Amir Khojastepour, Karthikeyan Sundaresan, Sampath Rangarajan, and Mung Chiang. Midu: Enabling mimo full duplex. In *MobiCom*, 2012.

[11] Sanaz Barghi, Amir Khojastepour, Karthik Sundaresan, and Sampath Rangarajan. Characterizing the throughput gain of single cell mimo wireless systems with full duplex radios. In *WiOpt*, 2012.

[12] Jona Beysens, Ander Galisteo, Qing Wang, Diego Juara, Domenico Giustiniano, and Sofie Pollin. Densevlc: a cell-free massive mimo system with distributed leds. In *CoNEXT*, 2018.

[13] Dinesh Bharadia, Kiran Raj Joshi, and Sachin Katti. Full duplex backscatter. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, 2013.

[14] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. Backfi: High throughput wifi backscatter. 2015.

[15] Dinesh Bharadia and Sachin Katti. Full duplex {MIMO} radios. In *NSDI*, 2014.

[16] Justin Chan, Anran Wang, Vikram Iyer, and Shyamnath Gollakota. Surface mimo: Using conductive surfaces for mimo between small devices. In *MobiCom*, 2018.

[17] Zicheng Chi, Yan Li, Xin Liu, Yao Yao, Yanchao Zhang, and Ting Zhu. Parallel inclusive communication for connecting heterogeneous iot devices at the edge. SenSys '19, 2019.

[18] Zicheng Chi, Yan Li, Hongyu Sun, Yao Yao, Zheng Lu, and Ting Zhu. B2w2: N-way concurrent communication for iot devices. SenSys '16.

[19] Junsu Choi, Sunghyun Choi, and Kwang Bok Lee. Sounding node set and sounding interval determination for ieee 802.11 ac mu-mimo. *IEEE Transactions on Vehicular Technology*, 2016.

[20] Wikipedia contributors. Moore–penrose inverse, 2019.

[21] Lara Deek, Eduard Garcia-Villegas, Elizabeth Belding, Sung-Ju Lee, and Kevin Almeroth. Joint rate and channel width adaptation for 802.11 mimo wireless networks. In *SECON*, 2013.

[22] Roberto Di Candia, Riku Jäntti, Ruifeng Duan, Jari Lietzén, Hany Khalifa, and Kalle Ruttik. Quantum backscatter communication: A new paradigm. In *ISWCS*, 2018.

[23] Lee Kai Fong and Luk Kwai Man. *Microstrip Patch Antennas*. World Scientific, 2001.

[24] Wei Gong, Haoxiang Liu, Kebin Liu, Qiang Ma, and Yunhao Liu. Exploiting channel diversity for rate adaptation in backscatter communication networks. In *IEEE INFOCOM*, 2016.

[25] Deke Guo, Yuan He, Yunhao Liu, Panlong Yang, Xiang-Yang Li, and Xin Wang. Link scheduling for exploiting spatial reuse in multihop mimo networks. *TPDS*, 2013.

[26] Ezzeldin Hamed, Hariharan Rahul, Mohammed A. Abdelghany, and Dina Katabi. Real-time distributed mimo systems. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16.

[27] Ezzeldin Hamed, Hariharan Rahul, and Bahar Partov. Chorus: Truly distributed distributed-mimo. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18.

[28] Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota. Netscatter: Enabling large-scale backscatter networks. NSDI'19.

[29] MM Aftab Hossain, Cicek Cavdar, Emil Björnson, and Riku Jäntti. Energy saving game for massive mimo: Coping with daily load variation. *IEEE Transactions on Vehicular Technology*, 2018.

[30] Vikram Iyer, Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *SIGCOMM 2016*, 2016.

[31] Kiran Joshi, Dinesh Bharadia, Manikanta Kotaru, and Sachin Katti. Wideo: Fine-grained device-free motion tracing using RF backscatter. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[32] Petteri Kela, Mário Costa, Jussi Turkka, Kari Leppänen, and Riku Jäntti. Flexible backhauling with massive mimo for ultra-dense networks. *IEEE Access*, 2016.

[33] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R. Smith, and David Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *SIGCOMM*, 2014.

[34] Bryce Kellogg, Vamsi Talla, Shyamnath Gollakota, and Joshua R. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *NSDI*, 2016.

[35] Tae Hyun Kim, Robert W Heath, and Sunghyun Choi. Multiuser mimo downlink with limited feedback using transmit-beam matching. In *ICC*, 2008.

[36] N. Kingsley, D. E. Anagnostou, M. Tentzeris, and J. Papapolymerou. Rf mems sequentially reconfigurable sierpinski antenna on a flexible organic substrate with novel dc-biasing technique. *Journal of Microelectromechanical Systems*, 2007.

[37] Yan Li, Zicheng Chi, Xin Liu, and Ting Zhu. Chiron: Concurrent high throughput communication for iot devices. MobiSys '18.

[38] Yan Li, Zicheng Chi, Xin Liu, and Ting Zhu. Passive-zigbee: Enabling zigbee communication in iot networks with 1000x+ less power consumption. SenSys '18.

[39] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *SIGCOMM*, 2013.

[40] Microsemi. *IGLOO nano Low Power Flash FPGAs with Flash Freeze Technology*, 2019. Revision 19.

[41] Saman Naderiparizi, Mehrdad Hessar, Vamsi Talla, Shyamnath Gollakota, and Joshua R Smith. Towards battery-free hd video streaming. In *NSDI*, 2018.

[42] P. V. Nikitin and K. V. S. Rao. Theory and measurement of backscattering from rfid tags. *IEEE Antennas and Propagation Magazine*, 2006.

[43] Konstantinos Nikitopoulos and Kyle Jamieson. Faster: Fine and accurate synchronization for large distributed mimo wireless networks. 2013.

[44] Konstantinos Nikitopoulos, Juan Zhou, Ben Congdon, and Kyle Jamieson. Geosphere: Consistently turning mimo capacity into throughput. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14.

[45] NVIDIA. Jetson nano brings ai computing to everyone. `https://devblogs.nvidia.com/jetson-nano-ai-computing/`, 2019.

[46] Ioannis Pefkianakis, Chi-Yu Li, Chunyi Peng, Suk-Bok Lee, and Songwu Lu. Cmes: Collaborative energy save for mimo 802.11 wireless networks. In *ICNP*, 2013.

[47] Yao Peng, Longfei Shangguan, Yue Hu, Yujie Qian, Xianshang Lin, Xiaojiang Chen, Dingyi Fang, and Kyle Jamieson. Plora: A passive long-range data network from ambient lora transmissions. In *SIGCOMM*, 2018.

[48] Wei-Liang Shen, Kate Ching-Ju Lin, Ming-Syan Chen, and Kun Tan. Client as a first-class citizen: Practical user-centric network mimo clustering. In *INFOCOM*, 2016.

[49] Wei-Liang Shen, Yu-Chih Tung, Kuang-Che Lee, Kate Ching-Ju Lin, Shyamnath Gollakota, Dina Katabi, and Ming-Syan Chen. Rate adaptation for 802.11 multiuser mimo networks. In *Mobicom*, 2012.

[50] Sanjib Sur, Ioannis Pefkianakis, Xinyu Zhang, and Kyu-Han Kim. Practical mu-mimo user selection on 802.11ac commodity networks. In *MobiCom*, 2016.

[51] Espressif Systems. *ESP8266EX Datasheet*. Espressif Systems, 2018.

[52] Deepak Vasisht, Guo Zhang, Omid Abari, Hsiao-Ming Lu, Jacob Flanz, and Dina Katabi. In-body backscatter communication and localization. In *SIGCOMM*, 2018.

[53] Anran Wang, Vikram Iyer, Vamsi Talla, Joshua R. Smith, and Shyamnath Gollakota. Fm backscatter: Enabling connected cities and smart fabrics. In *NSDI*, 2017.

[54] Wikipedia contributors. Orthogonal matrix — Wikipedia, the free encyclopedia, 2019. [Online; accessed 8-September-2019].

[55] Wikipedia contributors. Space–time block code — Wikipedia, the free encyclopedia, 2019. [Online; accessed 31-August-2019].

[56] Xiufeng Xie, Eugene Chai, Xinyu Zhang, Karthikeyan Sundaresan, Amir Khojastepour, and Sampath Rangarajan. Hekaton: Efficient and practical large-scale mimo. In *MobiCom*, 2015.

[57] Xiufeng Xie, Xinyu Zhang, and Karthikeyan Sundaresan. Adaptive feedback compression for mimo networks. In *MobiCom*, 2013.

[58] Gang Yang and Ying-Chang Liang. Backscatter communications over ambient ofdm signals: Transceiver design and performance analysis. In *GLOBECOM*, 2016.

[59] Zhice Yang, Qianyi Huang, and Qian Zhang. Nicscatter: Backscatter as a covert channel in mobile devices. In *MobiCom*, 2017.

[60] Yang-Seok Choi, P. J. Voltz, and F. A. Cassara. On channel estimation and detection for multicarrier signals in fast and selective rayleigh fading channels. *IEEE Transactions on Communications*, 2001.

[61] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. Hitchhike: Practical backscatter using commodity wifi. In *SenSys*, 2016.

[62] Pengyu Zhang, Colleen Josephson, Dinesh Bharadia, and Sachin Katti. Freerider: Backscatter communication using commodity radios. In *CoNEXT*, 2017.

[63] PENGYU ZHANG, Mohammad Rostami, Pan Hu, and Deepak Ganesan. Enabling practical backscatter communication for on-body sensors. In *SIGCOMM*, 2016.

[64] Pengyu Zhang, Mohammad Rostami, Pan Hu, and Deepak Ganesan. Enabling practical backscatter communication for on-body sensors. In *SIGCOMM*, 2016.

[65] X. Zhang, L. Zhong, and A. Sabharwal. Directional training for fdd massive mimo. *IEEE Transactions on Wireless Communications*, 2018.

[66] Xing Zhang, Lin Zhong, and Ashutosh Sabharwal. Directional training for fdd massive mimo. *IEEE Transactions on Wireless Communications*, 2018.

[67] Xinyu Zhang, Karthikeyan Sundaresan, Mohammad A. (Amir) Khojastepour, Sampath Rangarajan, and Kang G. Shin. Nemox: Scalable network mimo for wireless networks. In *MobiCom*, 2013.

[68] Jia Zhao, Wei Gong, and Jiangchuan Liu. Spatial stream backscatter using commodity wifi. In *MobiSys*, 2018.

[69] Jia Zhao, Wei Gong, and Jiangchuan Liu. X-tandem: Towards multi-hop backscatter communication with commodity wifi. In *MobiCom*, 2018.

Performant TCP for Low-Power Wireless Networks

Sam Kumar, Michael P Andersen, Hyung-Sin Kim, and David E. Culler
University of California, Berkeley

Abstract

Low-power and lossy networks (LLNs) enable diverse applications integrating many resource-constrained embedded devices, often requiring interconnectivity with existing TCP/IP networks as part of the Internet of Things. But TCP has received little attention in LLNs due to concerns about its overhead and performance, leading to LLN-specific protocols that require specialized gateways for interoperability. We present a systematic study of a well-designed TCP stack in IEEE 802.15.4-based LLNs, based on the TCP protocol logic in FreeBSD. Through careful implementation and extensive experiments, we show that modern low-power sensor platforms are capable of running full-scale TCP and that TCP, counter to common belief, performs well despite the lossy nature of LLNs. By carefully studying the interaction between the transport and link layers, we identify subtle but important modifications to both, achieving TCP goodput within 25% of an upper bound (5–40x higher than prior results) and low-power operation commensurate to CoAP in a practical LLN application scenario. This suggests that a TCP-based transport layer, seamlessly interoperable with existing TCP/IP networks, is viable and performant in LLNs.

1 Introduction

Research on wireless networks of low-power, resource constrained, embedded devices—low-power and lossy networks (LLNs) in IETF terms [128]—blossomed in the late 1990s. To obtain freedom to tackle the unique challenges of LLNs, researchers initially departed from the established conventions of the Internet architecture [50, 68]. As the field matured, however, researchers found ways to address these challenges *within* the Internet architecture [70]. Since then, it has become commonplace to use IPv6 in LLNs via the 6LoWPAN [105] adaptation layer. IPv6-based routing protocols, like RPL [33], and application-layer transports over UDP, like CoAP [35], have become standards in LLNs. Most wireless sensor network (WSN) operating systems, such as TinyOS [95], RIOT [24], and Contiki [45], ship with IP implementations enabled and configured. Major industry vendors offer branded and supported 6LoWPAN stacks (e.g., TI SimpleLink, Atmel SmartConnect). A consortium, Thread [64], has formed around 6LoWPAN-based interoperability.

Despite these developments, transport in LLNs has remained ad-hoc and TCP has received little serious consideration. Many embedded IP stacks (e.g., OpenThread [106]) do not even support TCP, and those that do implement only a subset of its features (Appendix B). The conventional wisdom is that IP holds merit, but *TCP is ill-suited to LLNs*. This view is represented by concerns about TCP, such as:

- "TCP is not light weight ... and may not be suitable for implementation in low-cost sensor nodes with limited processing, memory and energy resources." [110] (Similar argument in [42], [75].)
- That "TCP is a connection-oriented protocol" is a poor match for WSNs, "where actual data might be only in the order of a few bytes." [114] (Similar argument in [110].)
- "TCP uses a single packet drop to infer that the network is congested." This "can result in extremely poor transport performance because wireless links tend to exhibit relatively high packet loss rates." [109] (Similar argument in [43], [44], [75].)

Such viewpoints have led to a plethora of WSN-specialized protocols and systems [110, 117, 132] for reliable data transport, such as PSFQ [130], STCP [75], RCRT [109], Flush [88], RMST [125], Wisden [138], CRRT [10], and CoAP [31], and for unreliable data transport, like CODA [131], ESRT [118], Fusion [71], CentRoute [126], Surge [94], and RBC [142].

As LLNs become part of the emerging Internet of Things (IoT), it behooves us to re-examine the transport question, with attention to how the landscape has shifted: (1) As part of IoT, LLNs must be interoperable with traditional TCP/IP networks; to this end, using TCP in LLNs simplifies IoT gateway design. (2) Popular IoT application protocols, like MQTT [39] and ZeroMQ [8], assume that TCP is used at the transport layer. (3) Some IoT application scenarios demand high link utilization and reliability on low-bandwidth lossy links. Embedded hardware has also evolved substantially, prompting us to revisit TCP's overhead. In this context, **this paper seeks to determine: Do the "common wisdom" concerns about TCP hold in a modern IEEE 802.15.4-based LLN? Is TCP (still) unsuitable for use in LLNs?**

To answer this question, we leverage the fully-featured TCP implementation in the FreeBSD Operating System (rather than a limited locally-developed implementation) and refactor it to work with the Berkeley Low-Power IP Stack (BLIP), Generic Network Stack (GNRC), and OpenThread network stack, on two modern LLN platforms (§5). Naïvely running TCP in an LLN indeed results in poor performance. However, upon close examination, we discover that this is not caused by the expected reasons, such as those listed above. The *actual* reasons for poor TCP performance include (1) small link-layer frames that increase TCP header overhead, (2) hidden terminal effects over multiple wireless hops, and (3) poor interaction between TCP and a duty-cycled link. Through

Challenge	Technique	Observed Improvement
Resource Constraints	Zero-Copy Send	Send Buffer: 50% less mem.
	In-Place Reass.	Recv Buffer: 38% less mem.
Link-Layer Properties	Large MSS	TCP Goodput: 4–5x higher
	Link Retry Delay	TCP Seg. Loss: 6% $\to$ 1%
Energy Constraints	Adaptive DC	HTTP Latency: $\approx$ 2x lower
	L2 Queue Mgmt.	TCP Radio DC: 3% $\to$ 2%

Table 1: Impact of techniques to run full-scale TCP in LLNs

a systematic study of TCP in LLNs, we develop techniques to resolve these issues (Table 1), uncover why the generally assumed problems do not apply to TCP in LLNs, and show that TCP perfoms well in LLNs once these issues are resolved:

We find that **full-scale TCP fits well within the CPU and memory constraints of modern LLN platforms** (§5, §6). Owing to the low bandwidth of a low-power wireless link, a small window size ($\approx$ 2 KiB) is sufficient to fill the bandwidth-delay product and achieve good TCP performance. This translates into small send/receive buffers that fit comfortably within the memory of modern WSN hardware. Furthermore, we propose using an atypical Maximum Segment Size (MSS) to manage header overhead and packet fragmentation. As a result, **full-scale TCP operates well in LLNs, with 5–40 times higher throughput than existing (relatively simplistic) embedded TCP stacks** (§6).

Hidden terminals are a serious problem when running TCP over multiple wireless hops. We propose adding a delay d between link-layer retransmissions, and demonstrate that it effectively reduces hidden-terminal-induced packet loss for TCP. We find that, because a small window size is sufficient for good performance in LLNs, **TCP is quite resilient to spurious packet losses, as the congestion window can recover to a full window quickly after loss** (§7).

To run TCP in a low-power context, we *adaptively* duty-cycle the radio to avoid poor interactions with TCP's self-clocking behavior. We also propose careful link-layer queue management to make TCP more robust to interference. We demonstrate that **TCP can operate at low power, comparable to alternatives tailored specifically for WSNs**, and that **TCP brings value for real IoT sensor applications** (§8).

We conclude that TCP is entirely capable of running on IEEE 802.15.4 networks and low-cost embedded devices in LLN application scenarios (§9). Since our improvements to TCP and the link layer maintain seamless interoperability with other TCP/IP networks, we believe that a TCP-based transport architecture for LLNs could yield considerable benefit.

In summary, this paper's contributions are:

1. We implement a full-scale TCP stack for low-power embedded devices and reduce its resource usage.
2. We identify the actual issues causing poor TCP performance and develop techniques to address them.
3. We explain why the expected insurmountable reasons for poor TCP performance actually do not apply.
4. We demonstrate that, once these issues are resolved, TCP performs comparably to LoWPAN-specialized protocols.

Table 1 lists our techniques to run TCP in an LLN. Although prior LLN work has already used various forms of link-layer delays [136] and adaptive duty-cycling [140], our work shows, where applicable, how to adapt these techniques to work well with TCP, and demonstrates that they can address the challenges of LLNs within a *TCP-based* transport architecture.

2 Background and Related Work

Since the introduction of TCP, a vast literature has emerged, focusing on improving it as the Internet evolved. Some representative areas include congestion control [9, 51, 62, 76], performance on wireless links [15, 27], performance in high-bandwidth environments [11, 30, 53, 65, 78], mobility [124], and multipath operation [115]. Below, we discuss TCP in the context of LLNs and embedded devices.

2.1 Low-Power and Lossy Networks (LLNs)

Although the term *LLN* can be applied to a variety of technologies, including LoRa and Bluetooth Low Energy, we restrict our attention in this paper to **embedded networks using IEEE 802.15.4**. Such networks are called LoWPANs [93]—Low-Power Wireless Personal Area Networks—in contrast to WANs, LANs (802.3), and WLANs (802.11). Outside of LoWPANs, TCP has been successfully adapted to a variety of networks, including serial [77], Wi-Fi [27], cellular [25, 100], and satellite [15,25] links. While an 802.15.4 radio can in principle be added as a NIC to any device, we consider only *embedded* devices where it is the primary means of communication, running operating systems like TinyOS [68], RIOT [24], Contiki [45], or FreeRTOS. These devices are currently built around microcontrollers with Cortex-M CPUs, which lack MMUs. Below, we explain how LoWPANs are different from other networks where TCP has been successfully adapted.

Resource Constraints. When TCP was initially adopted by ARPANET in the early 1980s, contemporary Internet citizens—typically minicomputers and high-end workstations, but not yet personal computers—usually had at least 1 MiB of RAM. 1 MiB is tiny by today's standards, yet the LLN-class devices we consider in this work have *1-2 orders of magnitude less RAM than even the earliest computers connected with TCP/IP*. Due to energy constraints, particularly SRAM leakage, RAM size in low-power MCUs does not follow Moore's Law. For example, comparing Hamilton [83], which we use in this work, to TelosB [113], an LLN platform from 2004, shows only a 3.2x increase in RAM size over 16 years. This has caused LLN-class embedded devices to have a different balance of resources than conventional systems, a trend that is likely to continue well into the future. For example, whereas conventional computers have historically had roughly 1 MiB of RAM for every MIPS of CPU, as captured by the 3M rule, Hamilton has $\approx$ 50 DMIPS of CPU but only 32 KiB of RAM.

Link-Layer Properties. IEEE 802.15.4 is a low-bandwidth, wireless link with an MTU of only 104 bytes. The research community has explored using TCP with links that are *separately* low-bandwidth, wireless [27], or low-MTU [77], but addressing these issues *together* raises new challenges. For example, RTS-CTS, used in WLANs to avoid hidden terminals, has high overhead in LoWPANs [71, 136] due to the small MTU—control frames are comparable in size to data frames. Thus, LoWPAN researchers have moved away from RTS-CTS, instead carefully designing application traffic patterns to avoid hidden terminals [71, 88, 112]. Unlike Wi-Fi/LTE, LoWPANs do not use physical-layer techniques like adaptive modulation/coding or multi-antenna beamforming. Thus, they are directly impacted by link quality degradation due to varying environmental conditions [112, 127]. Additionally, IEEE 802.15.4 coexists with Wi-Fi in the 2.4 GHz frequency band, making Wi-Fi interference particularly relevant in indoor settings [99]. As LoWPANs are *embedded* networks, there is no human in the loop to react to and repair bad link quality.

Energy Constraints. Embedded nodes—the "hosts" of an LLN—are subject to strict power constraints. Low-power radios consume almost as much energy listening for a packet as they do when actually sending or receiving [20, 83]. Therefore, it is customary to *duty-cycle* the radio, keeping it in a low-power sleep state, in which it cannot send or receive data, most of the time [70, 112, 139]. The radio is only *occasionally* turned on to send/receive packets or determine if reception is likely. This requires *Media Management Control (MMC)* protocols [70, 112, 139] at the link layer to ensure that frames destined for a node are delivered to it only when its radio is on and listening. Similarly, the CPU also consumes a significant amount of energy [83], and must be kept idle most of the time.

Over the past 20 years, LLN researchers have addressed these challenges, but only in the context of special-purpose networks highly tailored to the particular application task at hand. The remaining open question is how to do so with a general-purpose reliable transport protocol like TCP.

2.2 TCP/IP for Embedded LLN-Class Devices

In the late 1990s and early 2000s, developers attempted to bring TCP/IP to embedded and resource-constrained systems to connect them to the Internet, usually over serial or Ethernet. Such systems [32, 80] were often designed with a specific application—often, a web server—in mind. These TCP/IP stacks were tailored to the specific applications at hand and were not suitable for general use. uIP ("micro IP") [42], introduced in 2002, was a standalone *general* TCP/IP stack optimized for 8-bit microcontrollers and serial or Ethernet links. To minimize resource consumption to run on such platforms, uIP omits standard features of TCP; for example, it allows only a single outstanding (unACKed) TCP segment per connection, rather than a sliding window of in-flight data.

Since the introduction of uIP, embedded networks have changed substantially. With *wireless* sensor networks and IEEE 802.15.4, various low-power networking protocols have been developed to overcome lossy links with strict energy and resource constraints, from S-MAC [139], B-MAC [112], X-MAC [34], and A-MAC [49], to Trickle [96] and CTP [59]. Researchers have viewed TCP as unsuitable, however, questioning end-to-end recovery, loss-triggered congestion control, and bi-directional data flow in LLNs [44]. Furthermore, WSNs of this era typically did not even use IP; instead, each WSN was designed specifically to support a particular application [89, 102, 138]. Those that require global connectivity rely on application-specific "base stations" or "gateways" connected to a TCP/IP network, treating the LLN like a peripheral interconnect (e.g., USB, bluetooth) rather than a network in its own right. This is because the prevailing sentiment at the time was that LLNs are too different from other types of networks and have to operate in too extreme conditions for the layered Internet architecture to be appropriate [50].

In 2007, the 6LoWPAN adaptation layer [105] was introduced, enabling IPv6 over IEEE 802.15.4. IPv6 has since been adopted in LLNs, bringing forth IoT [70]. uIP has been ported to LLNs [48], and IPv6 routing protocols, like RPL [33], and UDP-based application-layer transports, like CoAP [35], have emerged in LLNs. Representative operating systems, like TinyOS and Contiki, implement UDP/RPL/IPv6/6LoWPAN network stacks with IEEE 802.15.4-compatible MMC protocols for 16-bit platforms like TelosB [113].

TCP, however, is not widely adopted in LLNs. The few LLN studies that use TCP [47, 60, 67, 70, 72, 86, 144] generally use a simplified TCP stack (Appendix B), such as uIP.

In summary, despite the acceptance of IPv6, LLNs remain highly tailored at the transport layer to the application at hand. They typically use application-specific protocols on top of UDP; of such protocols, CoAP [31] has the widest adoption. In this context, this paper explores whether adopting TCP—and more broadly, the ecosystem of IP-based protocols, rather than IP alone—might bring value to LLNs moving forward.

3 Motivation: The Case for TCP in LLNs

As explained in §2, LLN design has historically been highly tailored to the specific application task at hand, for maximum efficiency. For example, PSFQ broadcasts data from a single source node to all others, RMST supports "directed diffusion" [73], and CoAP is tied to REST semantics. But embedded networks are not just isolated devices (e.g., peripheral interconnects like USB or bluetooth)—they are now true Internet citizens, and should be designed as such.

In particular, the recent megatrend of IoT requires LLNs to have a greater degree of *interoperability* with regular TCP/IP networks. Yet, LLN-specific protocols lack a clear separation between the transport and application layers, requiring *application-layer gateways* to communicate with TCP/IP-based services. This has encouraged IoT applications to develop as vertically-integrated silos, where devices cooperate only within an individual application or a particular manufacturer's ecosystem, with little to no interoperability *between* applications or with the general TCP/IP-based Internet. This phenomenon, sometimes called the "CompuServe of Things," is a serious obstacle to the IoT vision [57,97,104,133,141]. In contrast, other networks are seamlessly interoperable with the rest of the Internet. Accessing a new web application from a laptop does not require any new functionality at the Wi-Fi access point, but running a new application in a gateway-based LLN *does* require additional application-specific functionality to be installed at the gateway.

In this context, TCP-enabled LLN devices would be first-class citizens of the Internet, natively interoperable with the rest of the Internet via TCP/IP. They could use IoT protocols that assume a TCP-based transport layer (e.g., MQTT [39]) and security tools for TCP/IP networks (e.g., stateful firewalls), without an application-layer gateway. In addition, while traditional LLN applications like environment monitoring can be supported by unreliable UDP, certain applications do require high throughput and reliable delivery (e.g., anemometry (Appendix D), vibration monitoring [81]). TCP, *if it performs well in LLNs*, could benefit these applications.

Adopting TCP in LLNs may also open an interesting research agenda for IoT. TCP is the default transport protocol outside of LLNs, and history has shown that, to justify other transport protocols, application characteristics must offer substantial opportunity for optimization (e.g., [55,134,135]). If TCP becomes a viable option in LLNs, it would raise the bar for application-specific LLN protocols, resulting in some potentially interesting alternatives.

Although adopting TCP in LLNs could yield significant benefit and an interesting agenda, its feasibility and performance remain in question. This motivates our study.

4 Empirical Methodology

This section presents our methodology, carefully chosen to ground our study of full-scale TCP in LLNs.

4.1 Network Stack

Transport layer. That only a few full-scale TCP stacks exist, with a body of literature covering decades of refining, demonstrates that developing a feature-complete implementation of TCP is complex and error-prone [111]. Using a well-tested TCP implementation would ensure that results from our measurement study are due to the TCP *protocol*, not an artifact of the TCP *implementation* we used. Thus, we leverage the TCP implementation in FreeBSD 10.3 [56] to ground our study. We ported it to run in embedded operating systems and resource-constrained embedded devices (§4.2).

To verify the effectiveness of full-scale TCP in LLNs, we compare with CoAP [123], CoCoA [29], and unreliable UDP. CoAP is a standard LLN protocol that provides reliability on top of UDP. It is the most promising LLN alternative to TCP, gaining momentum in both academia [29,38,90,119,121,129] and industry [3,79], with adoption by Cisco [5,41], Nest/Google [4], and Arm [1,2]. CoCoA [29] is a recent proposal that augments CoAP with RTT estimation.

It is attractive to compare TCP to a variety of commercial systems, as has been done by a number of studies in LTE/WLANs [55,135]. Unfortunately, multihop LLNs have not yet reached the level of maturity to support a variety of commercial offerings; only CoAP has an appreciable level of commercial adoption. Other protocols are research proposals that often (1) are implemented for now-outdated operating systems and hardware or exist only in simulation [10,75,88], (2) target a very specific application paradigm [125,130,138], and/or (3) do not use IP [75,88,109,130]. We choose CoAP and CoCoA because they are not subject to these constraints.

	TelosB	Hamilton	Firestorm	Raspberry Pi
CPU	MSP430	Cortex-M0+	Cortex-M4	Cortex-A53
RAM	10 KiB	32 KiB	64 KiB	256 MB
ROM	48 KiB	256 KiB	512 KiB	SD Card

Table 2: Comparison of the platforms we used (Hamilton and Firestorm) to TelosB and Raspberry Pi

Layers 1 to 3. Because it is burdensome to place a border router with LAN connectivity within wireless range of every low-power host (e.g., sensor node), it is common to transmit data (e.g., readings) over *multiple* wireless LLN hops. Although each sensor must be battery-powered, it is reasonable to have a wall-powered LLN router node within transmission range of it.[1] This motivates Thread[2] [64,87], a recently developed protocol standard that constructs a multihop LLN over IEEE 802.15.4 links with *wall-powered, always-on* router nodes and *battery-powered, duty-cycled* leaf nodes. We use OpenThread [106], an open-source implementation of Thread.

Thread decouples routing from energy efficiency, providing a full-mesh topology among routers, frequent route updates, and asymmetric bidirectional routing for reliability. Each *leaf node* duty cycles its radio, and simply chooses a core router with good link quality, called its *parent*, as its next hop to all other nodes. The duty cycling uses *listen-after-send* [120]. A leaf node's parent stores downstream packets destined for that leaf node, until the leaf node sends it a *data request* message. A leaf node, therefore, can keep its radio powered off most of the time; infrequently, it sends a data request message to its parent, and turns on its radio for a short interval afterward to listen for downstream packets queued at its parent. Leaf nodes may send upstream traffic at any time. Each node uses CSMA-CA for medium access.

4.2 Embedded Hardware

We use two embedded hardware platforms: Hamilton [83] and Firestorm [18]. Hamilton uses a SAMR21 SoC with a 48 MHz Cortex-M0+, 256 KiB of ROM, and 32 KiB of RAM. Firestorm uses a SAM4L 48 MHz Cortex-M4 with 512 KiB of ROM and 64 KiB of RAM. While these platforms are more powerful than the TelosB [113], an older LLN platform widely

[1]The assumption of powered "core routers" is reasonable for most IoT use cases, which are typically indoors. Recent IoT protocols, such as Thread [64] and BLEmesh [63], take advantage of powered core routers.

[2]Thread has a large amount of industry support with a consortium already consisting of over 100 members [6], and is used in real IoT products sold by Nest/Google [7]. Given this trend, using Thread makes our work timely.

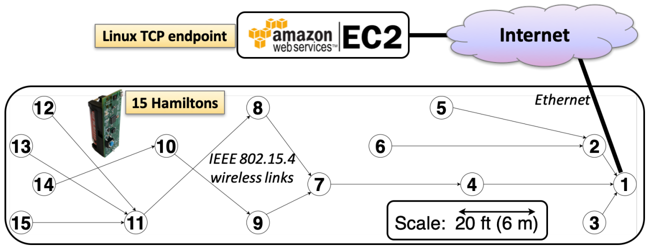


Figure 1: Snapshot of uplink routes in OpenThread topology at transmission power of -8 dBm (5 hops). Node 1 is the border router with Internet connectivity.

used in past studies, they are heavily resource-constrained compared to a Raspberry Pi (Table 2). Both platforms use the AT86RF233 radio, which supports IEEE 802.15.4. We use its standard data rate, 250 kb/s. We use Hamilton/OpenThread in our experiments; for comparison, we provide some results from Firestorm and other network stacks in Appendix A.

Handling automatic radio features. The AT86RF233 radio has built-in hardware support for link-layer retransmissions and CSMA-CA. However, it automatically enters low-power mode during CSMA backoff, during which it does not listen for incoming frames [20]. This behavior, which we call *deaf listening*, interacts poorly with TCP when radios are always on, because TCP requires bidirectional flow of packets—data in one direction and ACKs in the other. This may initially seem concerning, as deaf listening is an important power-saving feature. Fortunately, this problem disappears when using OpenThread's listen-after-send duty-cycling protocol, as leaf nodes never transmit data when listening for downstream packets. For experiments with always-on radios, we do not use the radio's capability for hardware CSMA and link retries; instead, we perform these operations in software.

Multihop Testbed. We construct an indoor LLN testbed, depicted in Figure 1, with 15 Hamiltons where node 1 is configured as the border router. OpenThread forms a 3-to-5-hop topology at transmission power of -8 dBm. Embedded TCP endpoints (Hamiltons) communicate with a Linux TCP endpoint (server on Amazon EC2) via the border router. During working hours, interference is present in the channel, due to people in the space using Wi-Fi and Bluetooth devices in the 2.4 GHz frequency band. At night, when there are few/no people in the space, there is much less interference.

5 Implementation of *TCPlp*

We seek to answer the following two questions: (1) Does full-scale TCP fit within the limited memory of modern LLN platforms? (2) How can we integrate a TCP implementation from a traditional OS into an embedded OS? To this end, we develop a TCP stack for LLNs based on the TCP implementation in FreeBSD 10.3, called *TCPlp* [91], on multiple embedded operating systems, RIOT OS [24] and TinyOS [95]. We use *TCPlp* in our measurement study in future sections.

Although we carefully preserved the protocol logic in the FreeBSD TCP implementation, achieving correct and performant operation on sensor platforms was a nontrivial effort. We had to modify the FreeBSD implementation according to the concurrency model of each embedded network stack and the timer abstractions provided by each embedded operating system (Appendix A). Our other modifications to FreeBSD, aimed at reducing memory footprint, are described below.

	Protocol	Socket Layer	`posix_sockets`
ROM	19972 B	6216 B	5468 B
RAM (Active)	364 B	88 B	48 B
RAM (Passive)	12 B	88 B	48 B

Table 3: Memory usage of *TCPlp* on RIOT OS. We also include RIOT's `posix_sockets` module, used by *TCPlp* to provide a Unix-like interface.

5.1 Connection State for *TCPlp*

As discussed in Appendix B, *TCPlp* includes features from FreeBSD that improve standard communication, like a sliding window, New Reno congestion control, zero-window probes, delayed ACKs, selective ACKs, TCP timestamps, and header prediction. *TCPlp*, however, omits some features in FreeBSD's TCP/IP stack. We omit dynamic window scaling, as buffers large enough to necessitate it ($\geq$ 64 KiB) would not fit in memory. We omit the urgent pointer, as it not recommended for use [61] and would only complicate buffering. Certain security features, such as host cache, TCP signatures, SYN cache, and SYN cookies are outside the scope of this work. We do, however, retain challenge ACKs [116].

We use separate structures for *active sockets* used to send and receive bytes, and *passive sockets* used to listen for incoming connections, as passive sockets require less memory.

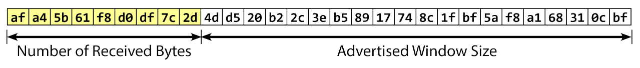
Table 3 depicts the memory footprint of *TCPlp* on RIOT OS. The memory required for the protocol and application state of an active TCP socket fits in a few hundred bytes, less than 1% of the available RAM on the Cortex-M4 (Firestorm) and 2% of that on the Cortex-M0+ (Hamilton). Although *TCPlp* includes heavyweight features not traditionally included in embedded TCP stacks, it fits well within available memory.

5.2 Memory-Efficient Data Buffering

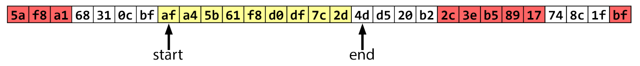
Existing embedded TCP stacks, such as uIP and BLIP, allow *only one TCP packet in the air*, eschewing careful implementation of send and receive buffers [86]. These buffers, however, are key to supporting TCP's sliding window functionality. We observe in §6.2 that *TCPlp* performs well with only 2-3 KiB send and receive buffers, which comfortably fit in memory even when naïvely pre-allocated at compile time. Given that buffers dominate *TCPlp*'s memory usage, however, we discuss techniques to optimize their memory usage.

5.2.1 Send Buffer: Zero-Copy

Zero-copy techniques [28, 40, 82, 98, 101] were devised for situations where the time for the CPU to copy memory is a significant bottleneck. Our situation is very different; the radio, not the CPU, is the bottleneck, owing to the low bandwidth of IEEE 802.15.4. By using a zero-copy send buffer,


af a4 5b 61 f8 d0 df 7c 2d 4d d5 20 b2 2c 3e b5 89 17 74 8c 1f bf 5a f8 a1 68 31 0c bf
Number of Received Bytes
Advertised Window Size

(a) Naïve receive buffer. Note that size of advertised window + size of buffered data = size of receive buffer.


5a f8 a1 68 31 0c bf af a4 5b 61 f8 d0 df 7c 2d 4d d5 20 b2 2c 3e b5 89 17 74 8c 1f bf
start
end

(b) Receive buffer with in-place reassembly queue. In-sequence data (yellow) is kept in a circular buffer, and out-of-order segments (red) are written in the space past the received data.

Figure 2: Naïve and final TCP receive buffers

however, we can avoid allocating memory to intermediate buffers that would otherwise be needed to copy data, thereby reducing the network stack's total memory usage.

In TinyOS, for example, the BLIP network stack supports vectored I/O; an outgoing packet passed to the IPv6 layer is specified as an `iovec`. Instead of allocating memory in the packet heap for each outgoing packet, *TCPlp* simply creates `iovec`s that point to existing data in the send buffer. This decreases the required size of the packet heap.

Unfortunately, zero-copy optimizations were not possible for the OpenThread implementation, because OpenThread does not support vectored I/O for sending packets. The result is that the *TCPlp* implementation requires a few kilobytes of additional memory for the send buffer on this platform.

5.2.2 Receive Buffer: In-Place Reassembly Queue

Not all zero-copy optimizations are useful in the embedded setting. In FreeBSD, received packets are passed to the TCP implementation as `mbufs` [137]. The receive buffer and reassembly buffer are `mbuf` chains, so data need not be copied out of `mbufs` to add them to either buffer or recover from out-of-order delivery. Furthermore, buffer sizes are chosen dynamically [122], and are merely a *limit* on their actual size. In our memory-constrained setting, such a design is dangerous because its memory usage is nondeterministic; there is additional memory overhead, due to headers, if the data are delivered in many small packets instead of a few large ones.

We opted for a flat array-based circular buffer for the receive buffer in *TCPlp*, primarily owing to its determinism in a limited-memory environment: buffer space is reserved at *compile-time*. Head/tail pointers delimit which part of the array stores in-sequence data. To reduce memory consumption, we store out-of-order data in the same receive buffer, at the same position as if they were in-sequence. We use a bitmap, not head/tail pointers, to record where out-of-order data are stored, because out-of-order data need not be contiguous. We call this an *in-place reassembly queue* (Figure 2).

6 TCP in a Low-Power Network

In this section, we characterize how full-scale TCP interacts with a low-power network stack, resource-constrained hardware, and a low-bandwidth link.

6.1 Reducing Header Overhead using MSS

In traditional networks, it is customary to set the Maximum Segment Size (MSS) to the link MTU (or path MTU) minus the size of the TCP/IP headers. IEEE 802.15.4 frames, however, are *an order of magnitude smaller* than frames in traditional networks (Table 4). The TCP/IP headers consume more than half of the frame's available MTU. As a result, TCP performs poorly, incurring more than 50% header overhead.

	Fast Ethernet	Wi-Fi	Ethernet	802.15.4
Capacity	100 Mb/s	54 Mb/s	10 Mb/s	250 kb/s
MTU	1500 B	1500 B	1500 B	104–116 B
Tx Time	0.12 ms	0.22 ms	1.2 ms	4.1 ms

Table 4: Comparison of TCP/IP links

Header	802.15.4	6LoWPAN	IPv6	TCP	Total
1st Frame	11–23 B	5 B	2–28 B	20–44 B	38–107 B
*n*th Frame	11–23 B	5–12 B	0 B	0 B	16–35 B

Table 5: Header overhead with 6LoWPAN fragmentation

Earlier approaches to running TCP over low-MTU links (e.g., low-speed serial links) have used TCP/IP header compression based on per-flow state [77] to reduce header overhead. In contrast, the 6LoWPAN adaptation layer [105], designed for LLNs, supports only *flow-independent* compression of the IPv6 header using shared link-layer state, a clear departure from per-flow techniques. A key reason for this is that the compressor and decompressor in an LLN (host and border router) are separated by several IP hops[3], making it desirable for intermediate nodes to be able to determine a packet's IP header without per-flow context (see §10 of [105]).

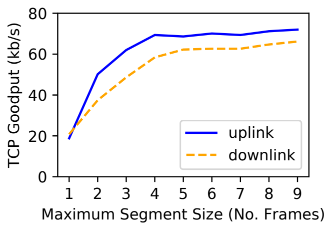
That said, compressing TCP headers separately from IP addresses using per-flow state is a promising approach to further amortize header overhead. There is preliminary work in this direction [22, 23], but it is based on uIP, which has one in-flight segment, and does not fully specify how to resynchronize compression state after packet loss with a multi-segment window. It is also not officially standardized by the IETF.

Therefore, this paper takes an approach orthogonal to header compression. We instead choose an MSS larger than the link MTU admits, *relying on fragmentation at the lower layers to decrease header overhead*. Fragmentation is handled by 6LoWPAN, which acts at Layer 2.5, between the link and network layers. Unlike end-to-end IP fragmentation, the 6LoWPAN fragments exist only within the LLN, and are reassembled into IPv6 packets when leaving the network.

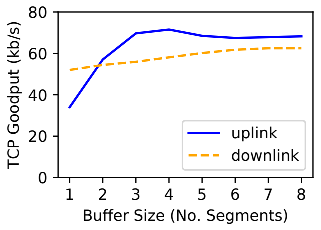
Relying on fragmentation is effective because, as shown in Table 5, TCP/IP headers consume space in the first fragment, but not in subsequent fragments. Using an excessively large MSS, however, decreases reliability because the loss of one fragment results in the loss of an entire packet. Existing work [21] has identified this trade-off and investigated it in simulation in the context of power consumption. We investigate it in the context of goodput in a live network.

Figure 3a shows the bandwidth as the MSS varies. As

[3]Thread deliberately does not abstract the mesh as a single IP link. Instead, it organizes the LLN mesh as a set of *overlapping link-local scopes*, using IP-layer routing to determine the path packets take through the mesh [70].

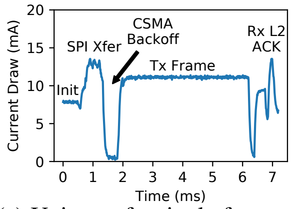


(a) Effect of varying MSS

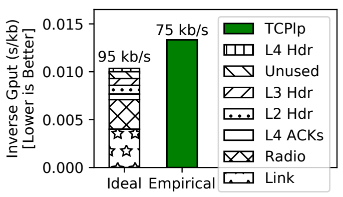


(b) Effect of varying buffer size

Figure 3: TCP goodput over one IEEE 802.15.4 hop



(a) Unicast of a single frame, measured with an oscilloscope



(b) *TCPlp* goodput compared with raw link bandwidth and overheads

Figure 4: Analysis of overhead limiting *TCPlp*'s goodput

expected, we see poor performance at a small MSS due to header overhead. Performance gains diminish when the MSS becomes larger than 5 frames. We recommend using an MSS of about 5 frames, but it is reasonable to decrease it to 3 frames if more wireless loss is expected. **Despite the small frame size of IEEE 802.15.4, we can effectively amortize header overhead for TCP using an atypical MSS.** Adjusting the MSS is orthogonal to TCP header compression. We hope that widespread use of TCP over 6LoWPAN, perhaps based on our work, will cause TCP header compression to be separately investigated and possibly used together with a large MSS.

6.2 Impact of Buffer Size

Whereas simple TCP stacks, like uIP, allow only one in-flight segment, full-scale TCP requires complex buffering (§5.2). In this section, we vary the size of the buffers (send buffer for uplink experiments and receive buffer for downlink experiments) to study how it affects the bandwidth. In varying the buffer size, we are directly affecting the size of TCP's flow window. We expect throughput to increase with the flow window size, with diminishing returns once it exceeds the bandwidth-delay product (BDP). The result is shown in Figure 3b. **Goodput levels off at a buffer size of 3 to 4 segments (1386 B to 1848 B), indicating that the buffer size needed to fill the BDP fits comfortably in memory.** Indeed, the BDP in this case is about $125\text{kb/s} \cdot 0.1\text{s} \approx 1.6\text{KiB}$.[4]

Downlink goodput at a buffer size of one segment is unusually high. This is because FreeBSD does not delay ACKs if the receive buffer is full, reducing the effective RTT from ≈ 130 ms to ≈ 70 ms. Indeed, goodput is very sensitive to RTT when the buffer size is small, because TCP exhibits "stop-and-wait" behavior due to the small flow window.

[4]We estimate the bandwidth as 125 kb/s rather than 250 kb/s to account for the radio overhead identified in §6.3.

6.3 Upper Bound on Single-Hop Goodput

We consider TCP goodput between two nodes over the IEEE 802.15.4 link, over a single hop without any border router. Using the Hamilton/OpenThread platform, we are able to achieve 75 kb/s.[5] Figure 4b lists various sources of overhead that limit *TCPlp*'s goodput, along with the ideal upper bounds that they admit. **Link** overhead refers to the 250 kb/s link capacity. **Radio** overhead includes SPI transfer to/from the radio (i.e., packet copying [107]), CSMA, and link-layer ACKs, which cannot be pipelined because the AT86RF233 radio has only one frame buffer. A full-sized 127-byte frame spends 4.1 ms in the air at 250 kb/s, but the radio takes 7.2 ms to send it (Figure 4a), almost halving the link bandwidth available to a single node. This is consistent with prior results [107]. **Unused** refers to unused space in link frames due to inefficiencies in the 6LoWPAN implementation. Overall, we estimate a 95 kb/s upper bound on goodput (100 kb/s without TCP headers). Our 75 kb/s measurement is within 25% of this upper bound, substantially higher than prior work (Table 6). The difference from the upper bound is likely due to network stack processing and other real-world inefficiencies.

7 TCP Over Multiple Wireless Hops

We instrument TCP connections between Hamilton nodes in our multi-hop testbed, without using the EC2 server.

7.1 Mitigating Hidden Terminals in LLNs

Prior work over traditional WLANs has shown that hidden terminals degrade TCP performance over multiple wireless hops [58]. Using RTS/CTS for hidden terminal avoidance has been shown to be effective in WLANs. This technique has an unacceptably high overhead in LLNs [136], however, because data frames are small (Table 4), comparable in size to the additional control frames required. Prior work in LLNs has carefully designed application traffic, using rate control [71, 88] and link-layer delays [136], to avoid hidden terminals.

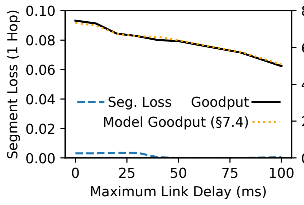
But prior work does not explore these techniques in the context of TCP. Unlike protocols like CoAP and simplified TCP implementations like uIP, a full-scale TCP flow has a *multi-segment sliding window* of unacknowledged data, making it unclear *a priori* whether existing LLN techniques will be sufficient. In particular, rate control seems sufficient because of bi-directional packet flow in TCP (data in one direction and ACKs in the other). So, rather than applying rate control, we attempt to avoid hidden terminals by adding a delay d between link-layer retries in addition to CSMA backoff. After a failed link transmission, a node waits for a random duration between 0 and d, before retransmitting the frame. The idea is

[5]Appendix A.4 provides the corresponding goodput figures for Hamilton/GNRC and Firestorm/BLIP platforms, for comparison.

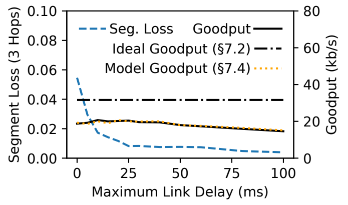
[6]One study [47] achieves ≈ 16 kb/s over multiple hops using the Linux TCP stack. We do not include it in Table 6 because it does not capture the resource constraints of LLNs—it uses traditional computers (PCs) for the end hosts—and does not consider hidden terminals—each hop uses a different wireless channel. It also uses TCP as a workload to evaluate a new link-layer protocol (burst forwarding), instead of evaluating TCP in its own right

	[144]	[22]	[67]	[86]	[69, 70]	This Paper (Hamilton Platform)
TCP Stack	uIP	uIP	uIP	BLIP	Arch Rock	*TCPlp* (RIOT OS, OpenThread)
Max. Seg Size	1 Frame	1 Frame	4 Frames	1 Frame	1024 bytes	5 Frames
Window Size	1 Seg.	1 Seg.	1 Seg.	1 Seg.	1 Seg.	1848 bytes (4 Seg.)
Goodput (One Hop)	1.5 kb/s	≈ 6.4 kb/s	≈ 12 kb/s	≈ 4.8 kb/s	15 kb/s	75 kb/s
Goodput (Multi-Hop)	≈ 0.55 kb/s	≈ 1.9 kb/s	≈ 12 kb/s	≈ 2.4 kb/s	9.6 kb/s	20 kb/s

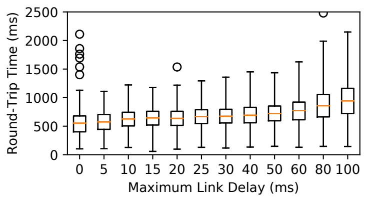
Table 6: Comparison of *TCPlp* to existing TCP implementations used in network studies over IEEE 802.15.4 networks.[6] Goodput figures obtained by reading graphs in the original paper (rather than stated numbers) are marked with the ≈ symbol.



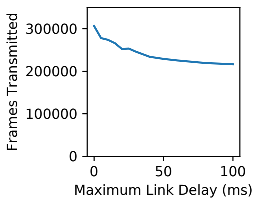
(a) TCP goodput, one hop



(b) TCP goodput, three hops



(c) RTT, three hops (outliers omitted)



(d) Total frames sent, three hops

Figure 5: Effect of varying time between link-layer retransmissions. Reported "segment loss" is the loss rate of TCP segments, not individual IEEE 802.15.4 frames. It includes only losses not masked by link-layer retries.

that if two frames collide due to a hidden terminal, the delay will prevent their link-layer retransmissions from colliding.

We modified OpenThread, which previously had no delay between link retries, to implement this. As expected, single-hop performance (Figure 5a) decreases as the delay between link retries increases; hidden terminals are not an issue in that setting. Packet loss is high for the multihop experiment (Figure 5b) when the link retry delay is 0, as is expected from hidden terminals. **Adding a small delay between link retries, however, effectively reduces packet loss.** Making the delay too large raises the RTT (Figure 5c).

We prefer a smaller frame/segment loss rate, even if goodput stays the same, in order to make more efficient use of network resources. Therefore, we prefer a moderate delay ($d = 40$ ms) to a small delay ($d = 5$ ms), even though both provide the same goodput, because the frame and segment loss rates are smaller when d is large (Figures 5b and 5d).

7.2 Upper Bound on Multi-Hop Goodput

Comparing Figures 5a and 5b, goodput over three wireless hops is substantially smaller than goodput over a single hop. Prior work has observed similar throughput reductions over multiple hops [86, 107]. It is due to radio scheduling constraints inherent in the multihop setting, which we describe in this section. Let B be the bandwidth over a single hop.

Consider a two-hop setup: $S \to R_1 \to D$. R_1 cannot receive a frame from S while sending a frame to D, because its radio cannot transmit and receive simultaneously. Thus, the maximum achievable bandwidth over two hops is $\frac{B}{2}$.

Now consider a three-hop setup: $S \to R_1 \to R_2 \to D$. By the same argument, if a frame is being transferred over $R_1 \to R_2$, then neither $S \to R_1$ nor $R_2 \to D$ can be active. Furthermore, if a frame is being transferred over $R_2 \to D$, then R_1 can hear that frame. Therefore, $S \to R_1$ cannot transfer a frame at that time; if it does, then its frame will collide at R_1 with the frame being transferred over $R_2 \to D$. Thus, the maximum bandwidth is $\frac{B}{3}$. We depict this ideal upper bound in Figure 5b, taking B to be the ideal single-hop goodput from §6.3.

In setups with more than three hops, every set of three adjacent hops is subject to this constraint. The first hop and fourth hop, however, may be able to transfer frames simultaneously. Therefore, the maximum bandwidth is still $\frac{B}{3}$. In practice, goodput may fall slightly because transmissions from a node may *interfere* with nodes multiple hops away, even if they can only be received by its immediate neighbors.

We made empirical measurements with $d = 40$ ms to validate this analysis. Goodput over one hop was 64.1 kb/s; over two hops, 28.3 kb/s; over three hops, 19.5 kb/s; and over four hops, 17.5 kb/s. This roughly fits the model.

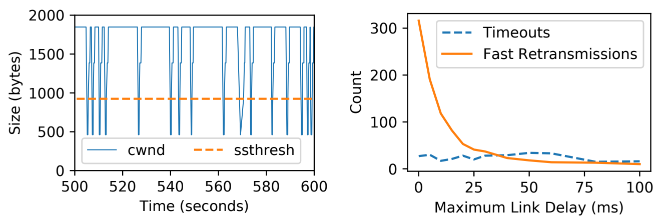
This analysis justifies why the same window size works well for both the one-hop experiments and the three-hop experiments in §7.1. Although the RTT is three times higher, the bandwidth-delay product is approximately the same. **Crucially, this means that the 2 KiB buffer size we determined in §6.2, which fits comfortably in memory, remains applicable for up to three wireless hops.**

7.3 TCP Congestion Control in LLNs

Recall that small send/receive buffers of only 1848 bytes (4 TCP segments) each are enough to achieve good TCP performance. This profoundly impacts TCP's congestion control mechanism. For example, consider Figure 5b. It is remarkable that throughput is almost the same at $d = 0$ ms and $d = 30$ ms, despite having 6% packet loss in the first case and less than 1% packet loss in the second.

Figure 6a depicts the congestion window over a 100 second interval during the $d = 0$ ms experiment.[7] Interestingly,

[7] All congestion events in Figure 6a were fast retransmissions, except for one timeout at $t = 569$ s. `cwnd` is temporarily set to 1 MSS during fast retransmissions due to an artifact of FreeBSD's implementation of SACK recovery. For clarity, we cap `cwnd` at the size of the send buffer, and we remove fluctuations in `cwnd` which resulted from "bad retransmissions" that the FreeBSD implementation corrected in the course of its normal execution.



(a) TCP cwnd for $d = 0$, three hops (b) TCP loss recovery, three hops

Figure 6: Congestion behavior of TCP over IEEE 802.15.4

the cwnd graph is far from the canonical sawtooth shape (e.g., Figure 11(b) in [26]); cwnd is almost always maxed out even though losses are frequent (6%). This is specific to small buffers. In traditional environments, where links have higher throughput and buffers are large, it takes longer for cwnd to recover after packet loss, greatly limiting the sending rate with frequent packet losses. In contrast, **in LLNs, where send/receive buffers are small, cwnd recovers to the maximum size quickly after packet loss, making TCP performance robust to packet loss.**

Congestion behavior also provides insight into loss patterns, as shown in Figure 6b. Fast retransmissions (used for isolated losses) become less frequent as d increases, suggesting that they are primarily caused by hidden-terminal-related losses. Timeouts do not become less frequent as d is increased, suggesting that they are caused by something else.

7.4 Modeling TCP Goodput in an LLN

Our findings in §7.3 suggest that, in LLNs, cwnd is limited by the buffer size, not packet loss. To validate this, we analytically model TCP performance according to our observations in §7.3, and then check if the resulting model is consistent with the data. Comprehensive models of TCP, which take window size limitations into account, already exist [108]; in contrast, our model is *intentionally simple* to provide intuition.

Observations in §7.3 suggest that we can neglect the time it takes the congestion window to recover after packet loss. So, we model a TCP connection as *binary*: either it is sending data with a full window, or it is not sending new data because it is recovering from packet loss. According to this model, a TCP flow alternates between *bursts* when it is transmitting at a full window, and *rests* when it is in recovery and not sending new data. Burst lengths depend on the packet loss rate p and rest lengths depend on RTT. This approach leads to the following model (full derivation is in Appendix C):

$$B = \frac{\text{MSS}}{\text{RTT}} \cdot \frac{1}{\frac{1}{w} + 2p} \qquad (1)$$

where B, the TCP goodput, is written in terms of the maximum segment size MSS, round-trip time RTT, packet loss rate p $(0 < p < 1)$, and window size w (sized to BDP, in packets). Figures 5a and 5b include the predicted goodput as dotted lines, calculated according to Equation 1 using the empirical RTT and segment loss rate for each experiment. **Our model of TCP goodput closely matches the empirical results.**

An established model of TCP outside of LLNs is [92, 103]:

$$B = \frac{\text{MSS}}{\text{RTT}} \cdot \sqrt{\frac{3}{2p}} \qquad (2)$$

Equation 2 fundamentally relies on there being many competing flows, so we do not expect it to match our empirical results from §7.3. But, given that existing work examining TCP in LLNs makes use of this formula to ground new algorithms [72], the differences between Equations 1 and 2 are interesting to study. In particular, Equation 1 has an added $\frac{1}{w}$ in the denominator and depends on p rather than $\sqrt{p}$, explaining, mathematically, how TCP in LLNs is more robust to small amounts of packet loss. We hope Equation 1, together with Equation 4 in Appendix C, will provide a foundation for future research on TCP in LLNs.

8 TCP in LLN Applications

To demonstrate that TCP is practical for real IoT use cases, we compare its performance to that of CoAP, CoCoA, and unreliable UDP in three workloads inspired by real application scenarios: web server, sense-and-send, and event detection. We evaluate the protocols over multiple hops with duty-cycled radios and wireless interference, present in our testbed in the day (§4.2). In our experiments, nodes 12–15 (Figure 1) send data to a server running on Amazon EC2. The RTT from the border router to the server was ≈ 12 ms, much smaller than within the low-power mesh (≈ 100-300 ms).

In our preliminary experiments, we found that in the presence of simultaneous TCP flows, tail drops at a relay node significantly impacted fairness. Implementing Random Early Detection (RED) [54] with Explicit Congestion Notification (ECN) support solved this problem. Therefore, we use RED and ECN for experiments in this section with multiple flows. While such solutions have sometimes been problematic since they are implemented in routers, they are more natural in LLNs because the intermediate “routers” relaying packets in an LLN typically also participate in the network as hosts.

We generally use a smaller MSS (3 frames) in this section, because it is more robust to interference in the day (§6). We briefly discuss how this affects our model in Appendix C, but leave a rigorous treatment to future work.

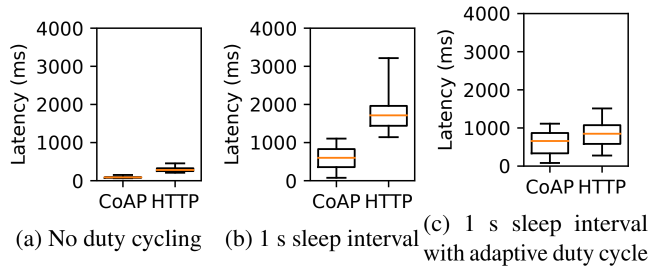
Running TCP in these application scenarios motivates **Adaptive Duty Cycle** and **Finer-Grained Link Queue Management**, which we introduce below as they are needed.

8.1 Web Server Application Scenario

To study TCP with multiple wireless hops and duty cycling, we begin with a web server hosted on a low-power device. We compare HTTP/TCP and CoAP/UDP (§4.1).

8.1.1 Latency Analysis

An HTTP request requires two round-trips: one to establish a TCP connection, and another for request/response. CoAP requires only one round trip (no connection establishment) and has smaller headers. Therefore, CoAP has a lower latency



(a) No duty cycling (b) 1 s sleep interval (c) 1 s sleep interval with adaptive duty cycle

Figure 7: Latency of web request: CoAP vs. HTTP/TCP

than HTTP/TCP when using an always-on link (Figure 7a). Even so, the latency of HTTP/TCP in this case is well below 1 second, not so large as to degrade user experience.

We now explore how a duty-cycled link affects the latency. Recall that leaf nodes in OpenThread (§4.1) periodically poll their parent to receive downstream packets, and keep their radios in a low-power sleep state between polls. We set the *sleep interval*—the time that a node waits between polls—to 1 s and show the latency in Figure 7b. Interestingly, HTTP's minimum observed latency is much higher than CoAP's, more than is explained by its additional round trip.

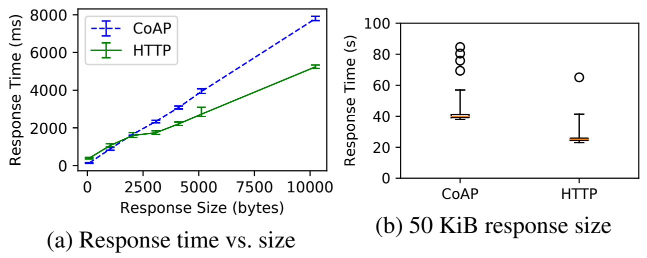
Upon investigation, we found that this is because **the self-clocking nature of TCP [76] interacts poorly with the duty-cycled link**. Concretely, the web server receives the SYN packet when it polls its parent, and sends the SYN-ACK immediately afterward, at the *beginning* of the next sleep interval. The web server therefore waits for the *entire* sleep interval before polling its parent again to receive the HTTP request, thereby experiencing the worst-case latency for the second round trip. We also observed this problem for batch transfer over TCP; TCP's self-clocking behavior causes it to consistently experience the worst-case round-trip time.

To solve this problem, we propose a technique called **Adaptive Duty Cycling**. After the web server receives a SYN, it *reduces the sleep interval* in anticipation of receiving an HTTP request. After serving the request, it restores the sleep interval to its old value. Unlike early LLN link-layer protocols like S-MAC [140] that use an adaptive duty cycle, we use *transport-layer state* to inform the duty cycle. Figure 7c shows the latency with adaptive duty cycling, where the sleep interval is temporarily reduced to 100 ms after connection establishment. **With adaptive duty-cycling, the latency overhead of HTTP compared to CoAP is small, despite larger headers and an extra round trip for connection establishment.**

Adaptive duty cycling is also useful in high-throughput scenarios, and in situations with persistent TCP connections. We apply adaptive duty cycling to one such scenario in §8.2.

8.1.2 Throughput Analysis

In §8.1.1, the size of the web server's response was 82 bytes, intentionally small to focus on latency. In a real application, however, the response may be large (e.g., it may contain a batch of sensor readings). In this section, we explore larger response sizes. We use a short sleep interval of 100 ms. This



(a) Response time vs. size (b) 50 KiB response size

Figure 8: Goodput: CoAP vs. HTTP/TCP

is realistic because, using adaptive duty cycling, the sleep interval may be longer when the node is idle, and reduced to 100 ms only when transferring the response.

Figure 8a shows the total time from dispatching the request to receiving the full response, as we vary the size of the response. It plots the median time, with quartiles shown in error bars. HTTP takes longer than CoAP when the response size is small (consistent with Figure 7), but CoAP takes longer when the response size is larger. This indicates that while HTTP/TCP has a greater fixed-size overhead than CoAP (higher y-intercept), it transfers data at a higher throughput (lower slope). TCP achieves a higher throughput than CoAP because CoAP sends response segments one-at-a-time ("stop and wait"), whereas TCP allows multiple segments to be in flight simultaneously ("sliding window").

To quantify the difference in throughput, we compare TCP and CoAP when transferring 50 KiB of data in Figure 8b. **TCP achieves 40% higher throughput compared to CoAP, over multiple hops and a duty-cycled link.**

8.1.3 Power Consumption

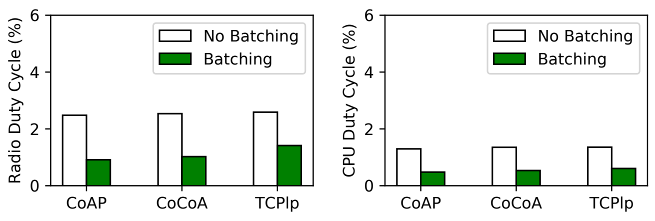
TCP consumes more energy than CoAP due to the extra round-trip at the beginning. In practice, however, a web server is interactive, and therefore will be *idle* most of the time. Thus, the idle power consumption dominates. For example, TCP keeps the radio on 35% longer than CoAP for a response size of 1024 bytes, but if the user makes one request every 100 seconds on average, this difference drops to only 0.35%.

Thus, we relegate in-depth power measurements to the sense-and-send application (§8.2), which is non-interactive.

8.2 Sense-and-Send Application Scenario

We turn our focus to the common *sense-and-send* paradigm, in which devices periodically collect sensor readings and send them upstream. For concreteness, we model our experiments on the deployment of anemometers in a building, a real-world LLN use case described in Appendix D. Anemometers collect measurements frequently (once per second), making heavy use of the transport protocol; given that our focus is on transport performance, this makes anemometers a good fit for our study. Other sensor deployments (e.g., temperature, humidity, building occupancy, etc.) sample data at a lower rate (e.g., 0.05 Hz), but are otherwise similar. Thus, *we expect our results to generalize to other sense-and-send applications.*

Nodes 12–15 (Figure 1) each generate one 82-byte reading every 1 second, and send it to the cloud server using either



(a) Radio duty cycle (b) CPU duty cycle

Figure 9: Effect of batching on power consumption

TCP or CoAP. We use most of the remaining RAM as an *application-layer queue* to prevent data from being lost if CoAP or TCP is in backoff after packet loss and cannot send out new data immediately. We make use of adaptive duty cycling for both TCP and CoAP, with a base sleep interval of four minutes (OpenThread's default) and decreasing it to 100 ms[8] when a TCP ACK or CoAP response is expected.

We measure a solution's *reliability* as the proportion of generated readings delivered to the server. Given that TCP and CoAP both guarantee reliability, a reliability measurement of less than 100% is caused by overflow of the application-layer queue due to poor network conditions preventing data from being reliably communicated as fast as they are generated. Generating data more slowly would result in higher reliability.

8.2.1 Performance in Favorable Conditions

We begin with experiments in our testbed at night, when there is less wireless interference. We compare three setups: (1) CoAP, (2) CoCoA, and (3) *TCPlp*. We also compare two sending scenarios: (1) sending each sensor reading right away ("No Batching"), and (2) sending sensor readings in batches of 64 ("Batching") [89]. We ensure that packets in a CoAP batch are the same size as segments in TCP (five frames).

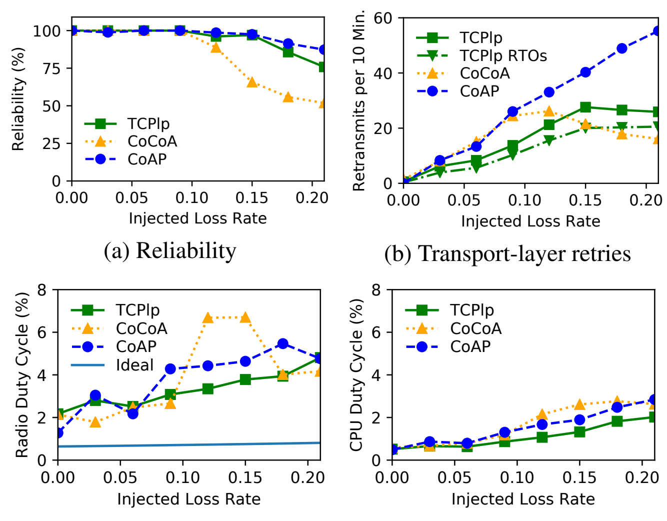
All setups achieved 100% reliability due to end-to-end acknowledgments (figures are omitted for brevity). Figures 9a and 9b also show that all the three protocols consume similar power; *TCP is comparable to LLN-specific solutions*.

Both the radio and CPU duty cycle are significantly smaller with batching than without batching. By sending data in batches, nodes can amortize the cost of sending data and waiting for a response. Thus, batching is the more realistic workload, so we use it to continue our evaluation.

8.2.2 Resilience to Packet Loss

In this section, we inject uniformly random packet loss at the border router and measure each solution.The result is shown in Figure 10. Note that the injected loss rate corresponds to the *packet-level* loss rate *after* link retries and 6LoWPAN reassembly. Although we plot loss rates up to 21%, *we consider loss rates > 15% exceptional; we focus on the loss rate up to 15%*. A number of WSN studies have already achieved > 90% end-to-end packet delivery, using only link/routing layer techniques (not transport) [46, 84, 85]. In our testbed environment, we have not observed the loss rate exceed 15% for an extended time, even with wireless interference.

[8] 100 ms is comparable to ContikiMAC's default sleep interval of 125 ms.



(a) Reliability (b) Transport-layer retries

(c) Radio duty cycle (d) CPU duty cycle

Figure 10: Performance with injected packet loss

Both CoAP and TCP achieve nearly 100% reliability at packet loss rates less than 15%, as shown in Figure 10a. At loss rates greater than 9%, CoCoA performs poorly. The reason is that CoCoA attempts to measure RTT for retransmitted packets, and conservatively calculates the RTT relative to the first transmission. This results in an inflated RTT value that causes CoCoA to delay longer before retransmitting, causing the application-layer queue to overflow. Full-scale TCP is immune to this problem despite measuring the RTT, because the TCP timestamp option allows TCP to unambiguously determine the RTT even for retransmitted segments.

Figures 10c and 10d show that, overall, **TCP and CoAP perform comparably in terms of radio and CPU duty cycle**. At 0% injected loss, *TCPlp* has a slightly higher duty cycle, consistent with Figure 9. At moderate packet loss, *TCPlp* appears to have a slightly lower duty cycle. This may be due to TCP's sliding window, which allows it to tolerate some ACK losses without retries. Additionally, Figure 10b shows that, although most of TCP's retransmissions are explained by timeouts, a significant portion were triggered in other ways (e.g., duplicate ACKs). In contrast, CoAP and CoCoA rely exclusively on timeouts, which has intrinsic limitations [143].

With exceptionally high packet loss rates (>15%), CoAP achieves higher reliability than TCP, because it "gives up" after just 4 retries; it exponentially increases the wait time between those retries, but then resets its RTO to 3 seconds when giving up and moving to the next packet. In contrast, TCP performs up to 12 retries with exponential backoff. Thus, TCP backs off further than CoAP upon consecutive packet losses, witnessed by the smaller retransmission count in Figure 10b, causing the application-layer queue to overflow more. This performance gap could be filled by parameter tuning.

We also consider an *ideal* "roofline" protocol to calculate a fairly loose lower bound on the duty cycle. This ideal protocol has the same header overhead as TCP, but learns which

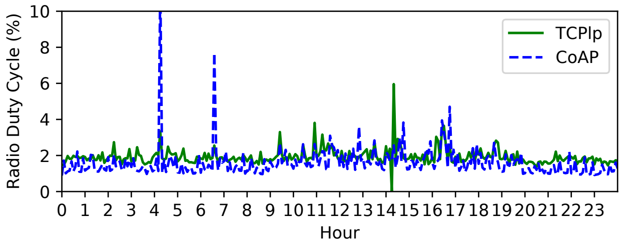


Figure 11: Radio duty cycle of TCP and CoAP in a lossy wireless environment, in one representative trial (losses are caused by natural human activity)

packets were lost for "free," without using ACKs or running MMC. Thus, it turns on its radio only to send out data and retransmit lost packets. The real protocols have much higher duty cycles than the ideal protocol would have (Figure 10c), suggesting that a significant amount of their overhead stems from determining which packets were lost—polling the parent node for downstream TCP ACKs/CoAP responses. This gap could be reduced by improving OpenThread's MMC protocol. For example, rather than using a fixed sleep interval of 100 ms when an ACK is expected, one could use exponential backoff to increase the sleep interval if an ACK is not quickly received. We leave exploring such ideas to future work.

8.2.3 Performance in Lossy Conditions

We compare the protocols over the course of a full day in our testbed, to study the impact of real wireless interference associated with human activity in an office. We focus on *TCPlp* and CoAP since they were the most promising protocols from the previous experiment. To ensure that *TCPlp* and CoAP are subject to similar interference patterns, we (1) run them simultaneously, and (2) hardcode adjacent *TCPlp* and CoAP nodes to have the same first hop in the multihop topology.

Improving Queue Management. OpenThread's queue management interacts poorly with TCP in the presence of interference. When a duty-cycled leaf node sends a data request message to its parent, it turns its radio on and listens until it receives a reply (called an "indirect message"). In OpenThread, the parent finishes sending its current frame (which may require link retries in the presence of interference), and then sends the indirect message. The duty-cycled leaf node keeps its radio on during this time, causing its radio duty cycle to increase. This is particularly bad for TCP, as its sliding window makes it more likely for the parent node to be in the middle of sending a frame when it receives a data request packet from a leaf node. Thus, **we modified OpenThread to allow indirect messages to preempt the current frame *in between link-layer retries***, to minimize the time that duty-cycled leaf nodes must wait for a reply with their radios on. Both TCP and CoAP benefitted from this; TCP benefitted more because it suffered more from the problem to begin with.

Power Consumption. To improve power consumption for both TCP and CoAP, we adjusted parameters according to the lossy environment: (1) we enabled link-layer retries for indirect messages, (2) we decreased the data request timeout and performed link-layer retries more rapidly for indirect messages, to deliver them to leaves more quickly, and (3) given the high level of daytime interference, we decreased the MSS from five frames to three frames (as in §8).

Protocol	Reliability	Radio DC	CPU DC
TCPlp	99.3%	2.29%	0.973%
CoAP	99.5%	1.84%	0.834%
Unrel., no batch	93.4%	1.13%	0.52%
Unrel., with batch	95.3%	0.734%	0.30%

Table 7: Performance in the testbed over a full day, averaged over multiple trials. The ideal protocol (§8.2.2) would have a radio DC of ≈ 0.63%–0.70% under similarly lossy conditions.

Figure 11 depicts the radio duty cycle of TCP and CoAP for a trial representative of our overall results. **CoAP maintains a lower duty cycle than *TCPlp* outside of working hours, when there is less interference; *TCPlp* has a slightly lower duty cycle than CoAP during working hours, when there is more wireless interference.** *TCPlp*'s better performance at a higher loss rate is consistent with our results from §8.2.2. At a lower packet loss rate, TCP performs slightly worse than CoAP. This could be due to hidden terminal losses; more retries, on average, are required for indirect messages for TCP, causing leaf nodes to stay awake longer. Overall, CoAP and *TCPlp* perform similarly (Table 7).

8.2.4 Unreliable UDP

As a point of comparison, we repeat the sense-and-send experiment using a UDP-based protocol that *does not provide reliability*. Concretely, we run CoAP in "nonconfirmable" mode, in which it does not use transport-layer ACKs or retransmissions. The result is in the last two rows of Table 7. Compared to unreliable UDP, reliable approaches increase the radio/CPU duty cycle by 3x, in exchange for nearly 100% reliability. That said, the corresponding decrease in battery life will be *less* than 3x, because other sources of power consumption (reading from sensors, idle current) are also significant.

For other sense-and-send applications that sample at a lower rate, TCP and CoAP would see higher reliability (less application queue loss), but UDP would not similarly benefit (no application queue). Furthermore, the power consumption of TCP, CoAP, and unreliable UDP would all be closer together, given that the radio and CPU spend more time idle.

8.3 Event Detection Application Scenario

Finally, we consider an application scenario where multiple flows compete for available bandwidth in an LLN. One such scenario is event detection: sensors wait until an interesting event occurs, at which point they report data upstream at a high data rate. Because such events tend to be correlated, multiple sensors send data simultaneously.

Nodes 12-15 in our testbed simultaneously transmit data to the EC2 instance (Figure 1), which measures the goodput

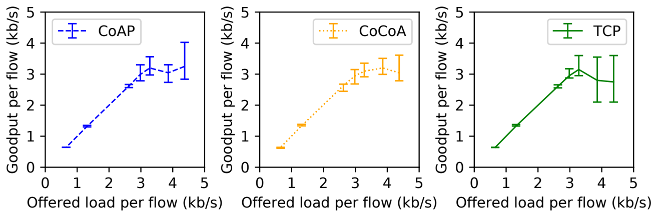


Figure 12: CoAP, CoCoA, and TCP with four competing flows

of each flow. We use the same duty-cycling policy as in §8.2. We divide each flow into 40-second intervals, measure the goodput in each interval, and compute the median and quartiles of goodput across all flows and intervals. The median gives a sense of aggregate goodput, and the quartiles gives a sense of fairness (quartiles close to the median are better).

Figure 12 shows the median and quartiles (as error bars) as the offered load increases. For small offered load, the per-flow goodput increases linearly. Once the aggregate load saturates the network, goodput declines slightly and the interquartile range increases, due to inefficiences in independent flows competing for bandwidth. **Overall, TCP performs similarly to CoAP and CoCoA, indicating that TCP's congestion control remains effective despite our observations in §7.3 that it behaves differently in LLNs.**

9 Conclusion

TCP is the *de facto* reliability protocol in the Internet. Over the past 40 years, new physical-, datalink-, and application-layer protocols have evolved alongside TCP, and supporting good TCP performance was a consideration in their design. TCP is the obvious performance baseline for new transport-layer proposals. To warrant adoption, novel transports must be *much* better than TCP in the intended application domain.

In contrast, when LLN research flourished two decades ago, LLN hardware could not run full-scale TCP. The original system architecture for networked sensors [68], for example, targeted an 8-bit MCU with only 512 *bytes* of memory. It naturally became taken for granted that TCP is too heavy for LLNs. Furthermore, contemporary research on TCP in WLANs [27] suggested that TCP would perform poorly in LLNs even if the resource constraints were surmounted.

In revisiting the TCP question, after the resource constraints relaxed, we find that the expected pitfalls of wireless TCP actually do not carry over to LLNs. Although naïve TCP indeed performs poorly in LLNs, this is not due to fundamental problems with TCP as were observed in WLANs. Rather, it is caused by incompatibilities with a low-power link layer, which likely arose because canonical LLN protocols were developed in the absence of TCP considerations. We show how to fix these incompatibilities while preserving seamless interoperability with other TCP/IP networks. This enables a viable TCP-based transport architecture for LLNs.

Our results have several implications for LLNs moving forward. First, **the use of lightweight protocols that emulate part of TCP's functionality, like CoAP, needs to be reconsidered.** Protocol stacks like OpenThread should support full-scale TCP as an option. TCP should also serve as a benchmark to assess new LLN transport proposals.

Second, **full-scale TCP will influence the design of networked systems using LLNs.** Such systems are presently designed with application-layer gateways in mind (§3). Using TCP/IP in the LLN itself would allow the use of commodity network management tools, like firewalls and NIDS. TCP would also allow the application-layer gateway to be replaced with a network-layer router, allowing clients to interact with LLN applications in much the same way as a Wi-Fi router allows users to interact with web applications. This is much more flexible than the status quo, where each LLN application needs application-specific functionality to be installed at the gateway [141]. In cases where a new LLN transport protocol is truly necessary, the new protocol may be wise to consider the byte-stream *abstraction* of TCP. This would allow the application-layer gateway to be replaced by a *transport-layer gateway*. The mere presence of a transport layer, distinct from the application layer, goes a long way to providing interoperability with the rest of the Internet.

Third, **UDP-based protocols will still have a place in LLNs, just as they have a place in the Internet.** UDP is used for applications that benefit from greater control of segment transmission and loss response than TCP provides. These are typically real-time or multimedia applications where losing information is preferable to late delivery. It is entirely seemly for some sensing applications in LLNs, particularly those with similar real-time constraints, to transfer data using UDP-based protocols, even if TCP is an option. But TCP *still* benefits such applications by providing a reliable channel for control information. For example, TCP may be used for device configuration, or to provide a shell for debugging, without yet another reliability protocol.

In summary, LLN-class devices are ready to become first-class citizens of the Internet. To this end, we believe that TCP should have a place in the LLN architecture moving forward, and that it will help put the "I" in IoT for LLN-class devices.

Acknowledgments

We thank the anonymous reviewers, including in prior submissions, and our shepherd, Keith Winstein, for their invaluable feedback. We are thankful to researchers in the BETS research group, including Kaifei Chen, Gabe Fierro, and Jack Kolb, for their feedback on early drafts of this paper, and to Prabal Dutta and Sylvia Ratnasamy for their advice and discussion. We also thank Albert Goto for his help with the LLN testbed.

This research is supported by the Department of Energy Grant DE-EE0007685, California Energy Commission, Intel Corporation, NSF Grant CPS-1239552, Fulbright Scholarship Program, UC Berkeley, and NSF Graduate Research Fellowship Program under Grant DGE-1752814. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Device management connect. https://www.arm.com/products/iot/pelion-iot-platform/device-management/connect. Accessed: 2018-09-09.

[2] Java speaks CoAP. https://community.arm.com/iot/b/blog/posts/java-speaks-coap. Accessed: 2018-09-09.

[3] MQTT and CoAP, IoT protocols. https://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php. Accessed: 2018-09-09.

[4] OpenThread. https://openthread.io/. Accessed: 2018-09-09.

[5] Software configuration guide, Cisco IOS release 15.2(5)ex (catalyst digital building series switches). https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst_digital_building_series_switches/software/15-2_5_ex/configuration_guide/b_1525ex_consolidated_cdb_cg/b_1525ex_consolidated_cdb_cg_chapter_0111101.html. Accessed: 2018-09-09.

[6] Thread group. https://www.threadgroup.org/thread-group#OurMembers. Accessed: 2018-09-11.

[7] What is Thread. https://www.threadgroup.org/What-is-Thread#threadready. Accessed: 2018-09-12.

[8] ZeroMQ. http://zeromq.org/. Accessed: 2019-01-29.

[9] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock. Host-to-host congestion control for TCP. *IEEE Communications Surveys & Tutorials*, 12(3), 2010.

[10] M. M. Alam and C. S. Hong. CRRT: congestion-aware and rate-controlled reliable transport in wireless sensor networks. *IEICE Transactions on Communications*, 92(1), 2009.

[11] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *SIGCOMM*. ACM, 2010.

[12] M. Allman. TCP byte counting refinements. *ACM SIGCOMM Computer Communication Review*, 29(3), 1999.

[13] M. Allman. TCP congestion control with appropriate byte counting (ABC). RFC 3465, 2003.

[14] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's loss recovery using limited transmit. RFC 3042, 2000.

[15] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP over satellite channels using standard mechanisms. RFC 2488, 1999.

[16] M. Allman and V. Paxson. On estimating end-to-end network path properties. *ACM SIGCOMM Computer Communication Review*, 29(4), 1999.

[17] M. Allman, V. Paxson, and E. Blanton. TCP congestion control. RFC 5681, 2009.

[18] M. P Andersen, G. Fierro, and D. E. Culler. System design for a synergistic, low power mote/BLE embedded platform. In *IPSN*. ACM/IEEE, 2016.

[19] E. Arens, A. Ghahramani, R. Przybyla, M. P Andersen, S. Min, T. Peffer, P. Raftery, M. Zhu, V. Luu, and H. Zhang. Measuring 3D indoor air velocity via an inexpensive low-power ultrasonic anemometer. *Energy and Buildings*, 211, 2020.

[20] Atmel Corporation. *Low Power, 2.4GHz Transceiver for ZigBee, RF4CE, IEEE 802.15.4, 6LoWPAN, and ISM Applications*, 2014. Preliminary Datasheet.

[21] A. Ayadi, P. Maillé, and D. Ros. TCP over low-power and lossy networks: tuning the segment size to minimize energy consumption. In *NTMS*. IEEE, 2011.

[22] A. Ayadi, P. Maillé, D. Ros, L. Toutain, and T. Zheng. Implementation and evaluation of a TCP header compression for 6LoWPAN. In *IWCMC*. IEEE, 2011.

[23] A. Ayadi, D. Ros, and L. Toutain. TCP header compression for 6LoWPAN: draft-aayadi-6lowpan-tcphc-01. Technical report, 2010. https://tools.ietf.org/id/draft-aayadi-6lowpan-tcphc-01.

[24] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. RIOT: an open source operating system for low-end embedded devices in the IoT. *IEEE Internet of Things Journal*, 2018.

[25] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The effects of asymmetry on TCP performance. In *MobiCom*. ACM, 1997.

[26] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, 5(6), 1997.

[27] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP performance over wireless networks. In *MobiCom*. ACM, 1995.

[28] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *SOSP*. ACM, 1989.

[29] A. Betzler, C. Gomez, I. Demirkol, and J. Paradells. CoAP congestion control for the Internet of Things. *IEEE Communications Magazine*, 54(7), 2016.

[30] D. Borman, B. Braden, and V. Jacobson. TCP extensions for high performance. (7323), 2014.

[31] C. Bormann, A. P. Castellani, and Z. Shelby. CoAP: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2), 2012.

[32] G. Borriello and R. Want. Embedded computation meets the world wide web. *Communications of the ACM*, 43(5), 2000.

[33] A. Brandt, J. W. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J.-P. Vasseur, and R. Alexander. RPL: IPv6 routing protocol for low-power and lossy networks. RFC 6550, 2012.

[34] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *SenSys*. ACM, 2006.

[35] A. P. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi. Web services for the Internet of Things through CoAP and EXI. In *ICC*. IEEE, 2011.

[36] D. D. Clark. The structuring of systems using upcalls. In *SOSP*. ACM, 1985.

[37] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications magazine*, 27(6), 1989.

[38] W. Colitti, K. Steenhaut, N. De Caro, B. Buta, and V. Dobrota. Evaluation of constrained application protocol for wireless sensor networks. In *LANMAN*. IEEE, 2011.

[39] MQTT Community. MQTT. http://mqtt.org. Accessed: January 25, 2018.

[40] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *SOSP*. ACM, 1993.

[41] P. Duffy. Beyond MQTT: A Cisco view on IoT protocols. https://blogs.cisco.com/digital/beyond-mqtt-a-cisco-view-on-iot-protocols. Accessed: 2018-09-09.

[42] A. Dunkels. Full TCP/IP for 8-bit architectures. In *MobiSys*. ACM, 2003.

[43] A. Dunkels, J. Alonso, and T. Voigt. Making TCP/IP viable for wireless sensor networks. *SICS Research Report*, 2003.

[44] A. Dunkels, J. Alonso, T. Voigt, H. Ritter, and J. Schiller. Connecting wireless sensornets with TCP/IP networks. In *International Conference on Wired/Wireless Internet Communications*. Springer, 2004.

[45] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN*. IEEE, 2004.

[46] S. Duquennoy, B. Al Nahas, O. Landsiedel, and T. Watteyne. Orchestra: Robust mesh networks through autonomously scheduled TSCH. In *SenSys*. ACM, 2015.

[47] S. Duquennoy, F. Österlind, and A. Dunkels. Lossy links, low power, high throughput. In *SenSys*. ACM, 2011.

[48] M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. Making sensor networks IPv6 ready. In *SenSys*. ACM, 2008.

[49] P. Dutta, S. Dawson-Haggerty, Y. Chen, C.-J. M. Liang, and A. Terzis. Design and evaluation of a versatile and efficient receiver-initiated link layer for low-power wireless. In *SenSys*. ACM, 2010.

[50] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *MobiCom*. ACM, 1999.

[51] K. Fall and S. Floyd. Simulation-based comparisons of tahoe, reno and SACK TCP. *ACM SIGCOMM Computer Communication Review*, 26(3), 1996.

[52] S. Floyd. TCP and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5), 1994.

[53] S. Floyd. HighSpeed TCP for large congestion windows. RFC 3649, 2003.

[54] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4), 1993.

[55] S. Fouladi, J. Emmons, E. Orbay, C. Wu, R. S. Wahby, and K. Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *NSDI*. USENIX, 2018.

[56] The FreeBSD Foundation. FreeBSD 10.3, 2016. https://www.freebsd.org/releases/10.3R/announce.html.

[57] J. Fürst, K. Chen, M. Aljarrah, and P. Bonnet. Leveraging physical locality to integrate smart appliances in non-residential buildings with ultrasound and bluetooth low energy. In *IoTDI*. IEEE, 2016.

[58] M. Gerla, K. Tang, and R. Bagrodia. TCP performance in wireless multi-hop networks. In *WMCSA*. IEEE, 1999.

[59] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *SenSys*. ACM, 2009.

[60] C. Gomez, A. Arcia-Moret, and J. Crowcroft. TCP in the Internet of Things: From ostracism to prominence. *IEEE Internet Computing*, 22(1), 2018.

[61] F. Gont and A. Yourtchenko. On the implementation of the TCP urgent mechanism. RFC 6093, 2011.

[62] L. A. Grieco and S. Mascolo. Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control. *ACM SIGCOMM Computer Communication Review*, 34(2), 2004.

[63] Bluetooth Mesh Working Group. Mesh profile v1.0, 2017.

[64] Thread Group. Thread, 2016. https://threadgroup.org.

[65] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5), 2008.

[66] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno modification to TCP's fast recovery algorithm. RFC 6582, 2012.

[67] K. Hewage, S. Duquennoy, V. Iyer, and T. Voigt. Enabling TCP in mobile cyber-physical systems. In *MASS*. IEEE, 2015.

[68] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS*. ACM, 2000.

[69] J. W. Hui. Personal Communication.

[70] J. W. Hui and D. E. Culler. IP is dead, long live IP for wireless sensor networks. In *SenSys*. ACM, 2008.

[71] B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating congestion in wireless sensor networks. In *SenSys*. ACM, 2004.

[72] H. Im. *TCP Performance Enhancement in Wireless Networks*. PhD thesis, Seoul National University, 2015.

[73] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCom*. ACM, 2000.

[74] D. Italiano and A. Motin. Calloutng: a new infrastructure for timer facilities in the FreeBSD kernel. In *AsiaBSDCon*, 2013.

[75] Y. G. Iyer, S. Gandham, and S. Venkatesan. STCP: a generic transport layer protocol for wireless sensor networks. In *ICCCN*. IEEE, 2005.

[76] V. Jacobson. Congestion avoidance and control. In *SIGCOMM*. ACM, 1988.

[77] V. Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, 1990.

[78] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: motivation, architecture, algorithms, performance. In *INFOCOM*. IEEE, 2004.

[79] S. Johnson. Constrained application protocol: CoAP is IoT's 'modern' protocol. https://www.omaspecworks.org/constrained-application-protocol-coap-is-iots-modern-protocol/, https://internetofthingsagenda.techtarget.com/feature/Constrained-Application-Protocol-CoAP-is-IoTs-modern-protocol. Accessed: 2018-09-09.

[80] H.-T. Ju, M.-J. Choi, and J. W. Hong. An efficient and lightweight embedded web server for web-based network element management. *International Journal of Network Management*, 10(5), 2000.

[81] D. Jung, Z. Zhang, and M. Winslett. Vibration analysis for IoT enabled predictive maintenance. In *ICDE*. IEEE, 2017.

[82] Yousef A. Khalidi and Moti N. Thadani. An efficient zero-copy I/O framework for UNIX. Technical report, Mountain View, CA, USA, 1995.

[83] H.-S. Kim, M. P Andersen, K. Chen, S. Kumar, W. J. Zhao, K. Ma, and D. E. Culler. System architecture directions for post-SoC/32-bit networked sensors. In *SenSys*. ACM, 2018.

[84] H.-S. Kim, H. Cho, H. Kim, and S. Bahk. DT-RPL: Diverse bidirectional traffic delivery through RPL routing protocol in low power and lossy networks. *Computer Networks*, 126, 2017.

[85] H.-S. Kim, H. Cho, M.-S. Lee, J. Paek, J. Ko, and S. Bahk. MarketNet: An asymmetric transmission power-based wireless system for managing e-price tags in markets. In *SenSys*. ACM, 2015.

[86] H.-S. Kim, H. Im, M.-S. Lee, J. Paek, and S. Bahk. A measurement study of TCP over RPL in low-power and lossy networks. *Journal of Communications and Networks*, 17(6), 2015.

[87] H.-S. Kim, S. Kumar, and D. E. Culler. Thread/OpenThread: A compromise in low-power wireless multihop network architecture for the Internet of Things. *IEEE Communications Magazine*, 57(7), 2019.

[88] S. Kim, R. Fonseca, P. Dutta, A. Tavakoli, D. E. Culler, P. Levis, S. Shenker, and I. Stoica. Flush: A reliable bulk transport protocol for multihop wireless networks. In *SenSys*. ACM, 2007.

[89] S. Kim, S. Pakzad, D. E. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *IPSN*. ACM/IEEE, 2007.

[90] M. Kovatsch, M. Lanter, and Z. Shelby. Californium: Scalable cloud services for the Internet of Things with CoAP. In *IOT*. IEEE, 2014.

[91] S. Kumar, M. P Andersen, H.-S. Kim, and D. E. Culler. Bringing full-scale TCP to low-power networks. In *SenSys*. ACM, 2018.

[92] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach*, chapter 3, pages 278–279. 6th edition, 2013.

[93] N. Kushalnagar, G. Montenegro, and C. Schumacher. IPv6 over low-power wireless personal area networks (6LoWPANs): Overview, assumptions, problem statement, and goals. RFC 4919, 2007.

[94] P. Levis, N. Lee, M. Welsh, and D. E. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *SenSys*. ACM, 2003.

[95] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. E. Culler. *TinyOS: An operating system for sensor networks*. 2005.

[96] P. Levis, N. Patel, D. E. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI*. USENIX, 2004.

[97] A. A. Levy, J. Hong, L. Riliskis, P. Levis, and K. Winstein. Beetle: Flexible communication for bluetooth low energy. In *MobiSys*. ACM, 2016.

[98] Y.-C. Li and M.-L. Chiang. LyraNET: a zero-copy TCP/IP protocol stack for embedded operating systems. In *RTCSA*. IEEE, 2005.

[99] C.-J. M. Liang, N. B. Priyantha, J. Liu, and A. Terzis. Surviving Wi-Fi interference in low power ZigBee networks. In *SenSys*. ACM, 2010.

[100] R. Ludwig, A. Gurtov, and F. Khafizov. TCP over second (2.5G) and third (3G) generation wireless networks. RFC 3481, 2003.

[101] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *SOSP*. ACM, 1993.

[102] A. Mainwaring, D. E. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*. ACM, 2002.

[103] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3), 1997.

[104] A. McEwen. Risking a compuserve of things. https://mcqn.com/posts/wuthering-bytes-slides-risking-a-compuserve-of-things/. Accessed: 2018-12-08.

[105] G. Montenegro, N. Kushalnagar, J. W. Hui, and D. E. Culler. Transmission of IPv6 packets over IEEE 802.15.4 networks. RFC 4944, 2007.

[106] Google Nest. OpenThread, 2017. https://github.com/openthread/openthread.

[107] F. Österlind and A. Dunkels. Approaching the maximum 802.15.4 multi-hop throughput. In *HotEmNets*. ACM, 2008.

[108] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. *ACM SIGCOMM Computer Communication Review*, 28(4), 1998.

[109] J. Paek and R. Govindan. RCRT: Rate-controlled reliable transport for wireless sensor networks. In *SenSys*. ACM, 2007.

[110] Q. Pang, V. W. S. Wong, and V. C. M. Leung. Reliable data transport and congestion control in wireless sensor networks. *International Journal of Sensor Networks*, 3(1), 2008.

[111] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP implementation problems. RFC 2525, 1999.

[112] J. Polastre, J. Hill, and D. E. Culler. Versatile low power media access for wireless sensor networks. In *SenSys*. ACM, 2004.

[113] J. Polastre, R. Szewczyk, and D. E. Culler. Telos: Enabling ultra-low power wireless research. In *IPSN*. ACM/IEEE, 2005.

[114] M. A. Rahman, A. El Saddik, and W. Gueaieb. *Wireless Sensor Network Transport Layer: State of the Art*. 2008.

[115] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? Designing and implementing a deployable multipath TCP. In *NSDI*. USENIX, 2012.

[116] A. Ramaiah, M. Stewart, and M. Dalal. Improving TCP's robustness to blind in-window attacks. RFC 5961, 2010.

[117] A. J. D. Rathnayaka and V. M. Potdar. Wireless sensor network transport protocol: A critical review. *Journal of Network and Computer Applications*, 36(1), 2013.

[118] Y. Sankarasubramaniam, Ö. B. Akan, and I. F. Akyildiz. Esrt: event-to-sink reliable transport in wireless sensor networks. In *MobiHoc*. ACM, 2003.

[119] D. F. S. Santos, H. O. Almeida, and A. Perkusich. A personal connected health system for the Internet of Things based on the Constrained Application Protocol. *Computers & Electrical Engineering*, 44, 2015.

[120] T. Schmid, R. Shea, M. B. Srivastava, and P. Dutta. Disentangling wireless sensing from mesh networking. In *HotEmNets*, 2010.

[121] K. Seitz, S. Serth, K.-F. Krentz, and C. Meinel. Enabling en-route filtering for end-to-end encrypted CoAP messages. In *SenSys*. ACM, 2017.

[122] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP buffer tuning. In *SIGCOMM*. ACM, 1998.

[123] Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (CoAP). RFC 7252, 2014.

[124] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MobiCom*. ACM, 2000.

[125] F. Stann and J. Heidemann. RMST: Reliable data transport in sensor networks. In *SNPA*. IEEE, 2003.

[126] T. Stathopoulos, L. Girod, J. Heidemann, and D. Estrin. Mote herding for tiered wireless sensor networks. Technical Report 58, University of California, Los Angeles, Center for Embedded Networked Computing, December 2005.

[127] R. Szewczyk, J. Polastre, A. Mainwaring, and D. E. Culler. Lessons from a sensor network expedition. In H. Karl, A. Wolisz, and A. Willig, editors, *EWSN*. Springer Berlin Heidelberg, 2004.

[128] J.-P. Vasseur. Terms used in routing for low-power and lossy networks. RFC 7102, 2014.

[129] B. C. Villaverde, D. Pesch, R. De Paz Alberola, S. Fedor, and M. Boubekeur. Constrained Application Protocol for low power embedded networks: A survey. In *IMIS*. IEEE, 2012.

[130] C.-Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: a reliable transport protocol for wireless sensor networks. In *WSNA*. ACM, 2002.

[131] C.-Y. Wan, S. B. Eisenman, and A. T. Campbell. CODA: Congestion detection and avoidance in sensor networks. In *SenSys*. ACM, 2003.

[132] C. Wang, K. Sohraby, Y. Hu, B. Li, and W. Tang. Issues of transport control protocols for wireless sensor networks. In *ICCCAS*. IEEE, 2005.

[133] P. Windley. The compuserve of things. `http://www.windley.com/archives/2014/04/the_compuserve_of_things.shtml`. Accessed: 2018-12-08.

[134] K. Winstein and H. Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *ATC*. USENIX, 2012.

[135] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*. USENIX, 2013.

[136] A. Woo and D. E. Culler. A transmission control scheme for media access in sensor networks. In *MobiCom*. ACM, 2001.

[137] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated*, volume 2, chapter 2. 1995.

[138] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *SenSys*. ACM, 2004.

[139] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *INFOCOM*. IEEE, 2002.

[140] W. Ye, J. Heidemann, and D. Estrin. Medium access control with coordinated adaptive sleeping for wireless sensor networks. *IEEE/ACM Transactions on Networking*, 12(3), 2004.

[141] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, and P. Dutta. The Internet of Things has a gateway problem. In *HotMobile*. ACM, 2015.

[142] H. Zhang, A. Arora, Y.-R. Choi, and M. G. Gouda. Reliable bursty convergecast in wireless sensor networks. In *MobiHoc*. ACM, 2005.

[143] L. Zhang. Why TCP timers don't work well. *ACM SIGCOMM Computer Communication Review*, 16(3), 1986.

[144] T. Zheng, A. Ayadi, and X. Jiang. TCP over 6LoWPAN for industrial applications: An experimental study. In *NTMS*. IEEE, 2011.

A Impact of Network Stack Design

As mentioned in §5, we made nontrivial modifications to FreeBSD's TCP stack to port it to each embedded operating system and embedded network stack. Below we provide additional information about these changes, and about our implementations for platforms other than Hamilton/OpenThread.

A.1 Concurrency Model

GNRC and OpenThread (RIOT OS). RIOT OS provides threads as the basic unit of concurrency. Asynchronous interaction with hardware is done by interrupt handlers that preempt the current thread, perform a short operation in the interrupt context, and signal a related thread to perform any remaining operation outside of interrupt context. Then the thread is placed on the RIOT OS scheduler queue and is scheduled for execution depending on its priority.

The GNRC network stack for RIOT OS runs each network layer (or module) in a separate thread. Each thread has a priority and can be preempted by a thread with higher priority or by an interrupt. The thread for a lower network layer has higher priority than the thread for a higher layer.

The port of OpenThread for RIOT OS handles received packets in one thread and sends packets from another thread, where the thread for received packets has higher priority [83]. The rationale for this design is to ensure timely processing of received packets at the radio, which is especially important in the context of a high-throughput flow.

To adapt *TCPlp* for GNRC, we run the FreeBSD implementation as a single TCP-layer thread, whose priority is between that of the application-layer thread and the IPv6-layer thread. To adapt *TCPlp* for OpenThread on RIOT OS, we call the TCP protocol logic (`tcp_input()`) at the appropriate point along the receive path, and send packets from the TCP protocol logic (`tcp_output()`) using the established send path. As explained in Appendix A.2, we also use an additional thread for timer callbacks in RIOT OS.

Given that TCP state can be accessed concurrently from multiple threads—the TCP thread (GNRC) or receive thread (OpenThread), the application thread(s), and timer callbacks—we needed to synchronize access to it. The FreeBSD implementation allows fine-grained locking of connection state to allow different connections to be serviced in parallel on different CPUs. Given that low-power embedded sensors typically have only one CPU, however, we opted for simplicity, instead using a single global TCP lock for *TCPlp*.

BLIP (TinyOS). TinyOS uses an event-driven concurrency model based on split-phase operations, consisting of an event loop that executes on a *single* stack. For concurrency, TinyOS provides three types of unique operations: *commands* and *events*, which are executed immediately, and *tasks*, which are scheduled for execution after all preceding tasks are completed. An interrupt handler may preempt the current function, perform a short operation in the interrupt context using *asynchronous* events and commands, and *post* a task to perform any remaining computation later. To adapt the thread-based FreeBSD implementation to the event-driven TinyOS, we execute the primary functions of FreeBSD, such as `tcp_output()` and `tcp_input()`, within *tasks* outside of interrupt context. Because tasks in TinyOS cannot preempt each other, we remove the locking present in the FreeBSD TCP implementation.

A.2 Timer Event Management

Given that many TCP operations are based on timer events, achieving correct timer operation is important. For example, if an RTO timer event is dropped by the embedded operating system, the RTO timer will not be rescheduled, and the connection may hang.

For a simple and stable operation, many existing embedded TCP stacks, including the uIP, lwIP, and BLIP TCP stacks, rely on a periodic, fixed-interval clock in order to check for expired timeouts. Instead, *TCPlp* uses one-shot tickless timers as FreeBSD 10.3 does [74], which is beneficial in two ways: (1) When there are no scheduled timers, the tickless timers allow the CPU to sleep, rather than being needlessly woken up at a fixed interval, resulting in lower energy consumption [83]. (2) Unlike fixed periodic timers, which can only be serviced on the next tick after they expire, tickless timers can be serviced as soon as they expire. To obtain these advantages, however, an embedded operating system must robustly manage asynchronous timer callbacks.

TinyOS has a single event queue maintained by the scheduler. The semantics of TinyOS guarantee that a task can exist in the event queue only once, even if it is *posted* (i.e., scheduled for execution) multiple times before executing. Therefore, the event queue can be sized appropriately at compile-time to not overflow. Furthermore, TinyOS handles received packets in a separate queue than tasks. This ensures that TCP

	Protocol	Event Sched.	User Library
ROM	21352 B	1696 B	5384 B
RAM (Active)	488 B	40 B	36 B
RAM (Passive)	16 B	16 B	36 B

Table 8: Memory usage of *TCPlp* on TinyOS. Our implementation of *TCPlp* spans three modules: (1) protocol implementation, (2) event scheduler that injects callbacks into userspace, and (3) userland library.

timer callbacks will not be dropped.

This is not the case for RIOT OS. Timer callbacks either handle the timer entirely in interrupt context, or put an event on a thread's message queue, so that the thread performs the required callback operation. Each network protocol supported by RIOT OS has a single thread. Because a thread's message queue in RIOT OS is used to hold both received packets and timer events, there is no guarantee when a timer expires that there is enough space in the thread message queue to accept a timer event; if there is not enough space, RIOT OS drops the timer event. Furthermore, if a timer expires multiple times before its event is handled by the thread, multiple events for the same timer can exist simultaneously in the queue; *we cannot find an upper bound on the number of slots in the message queue used by a single timer*. To provide robust TCP operation on RIOT OS, we create a second thread used exclusively for TCP timers. We handle timers similarly to TinyOS' *post* operation, by preventing the message queue from having multiple callback events of a single timer. This eliminates the possibility of timer event drops.

A.3 Memory Usage: Connection State

To complement Table 3, which shows *TCPlp*'s memory footprint on RIOT OS, we include Table 8, which shows *TCPlp*'s memory footprint on TinyOS.

A.4 Performance Comparison

We consider TCP goodput between two embedded nodes over the IEEE 802.15.4 link, over a single hop without any border router, as we did in §6.3. We are able to produce a 63 kb/s goodput over a TCP connection between two Hamilton motes using RIOT's GNRC network stack. For comparison, we are able to achieve 71 kb/s using the BLIP stack on Firestorm, and 75 kb/s using the OpenThread network stack with RIOT OS on Hamilton. **This suggests that our results are reproducible across multiple platforms and embedded network stacks.** The minor performance degradation in GNRC is partially explained by its greater header overhead due to implementation differences, and by its IPC-based thread-per-layer concurrency architecture, which has known inefficiencies [36]. This suggests that the implementation of the underlying network stack, particularly with regard to concurrency, could affect TCP performance in LLNs.

	uIP	BLIP	GNRC	*TCPlp*
Flow Control	Yes	Yes	Yes	Yes
Congestion Control	N/A	No	Yes	Yes
RTT Estimation	Yes	No	Yes	Yes
MSS Option	Yes	No	Yes	Yes
OOO Reassembly	No	No	Yes	Yes
TCP Timestamps	No	No	No	Yes
Selective ACKs	No	No	No	Yes
Delayed ACKs	No	No	No	Yes

Table 9: Comparison of core features among embedded TCP stacks: uIP (Contiki), BLIP (TinyOS), GNRC (RIOT), and *TCPlp (this paper)*

B Comparison of Features in Embedded TCP Implementations

Table 9 compares the featureset of *TCPlp* to features in embedded TCP stacks. The TCP implementations in uIP and BLIP lack features core to TCP. uIP allows only one unACKed in-flight segment, eschewing TCP's sliding window. BLIP does not implement RTT estimation or congestion control. The TCP implementation in GNRC lacks features such as TCP timestamps, selective ACKs, and delayed ACKs, which are present in most full-scale TCP implementations.

Benefits of full-scale TCP. In addition to supporting the protocol-level features summarized in Table 9, *TCPlp* is likely more robust than other embedded TCP stacks because it is based on a well-tested TCP implementation. While seemingly minor, some details, implemented incorrectly by TCP stacks, have had important consequences for TCP's behavior [111]. *TCPlp* benefits from a thorough implementation of each aspect of TCP.

For example, *TCPlp*, by virtue of using the FreeBSD TCP implementation, benefits from a robust implementation of congestion control. *TCPlp* implements not only the basic New Reno algorithm, but also Explicit Congestion Notification [52], Appropriate Byte Counting [12, 13] and Limited Transmissions [14]. It also inherits from FreeBSD heuristics to identify and correct "bad retransmissions" (as in §2.8 of [16]): if, after a retransmission, the corresponding ACK is received very soon (within $\frac{\text{RTT}}{2}$ of the retransmission), the ACK is assumed to correspond to the originally transmitted segment as opposed to the retransmission. The FreeBSD implementation and *TCPlp* recover from such "bad retransmissions" by restoring `cwnd` and `ssthresh` to their former values before the packet loss. Aside from congestion control, *TCPlp* benefits from header prediction [37], which introduces a "fast code path" to process common-case TCP segments (in-sequence data and ACKs) more efficiently, and Challenge ACKs [116], which make it more difficult for an attacker to inject an RST into a TCP connection.

Enhancements such as these make us more confident that our observed results are fundamental to TCP, as opposed to artifacts of poor implementation. Furthermore, they allow us to focus on performance problems arising from the challenges of LLNs, as opposed to general TCP-related challenges that the research community has already solved in the context of traditional networks and operating systems.

C Derivation of TCP Model

This appendix provides the derivation of Equation 1, the model of TCP performance proposed in §7.4.

We think of a TCP flow as a sequence of bursts. A *burst* is a sequence of full windows of data successfully transferred, which ends in a packet loss. After this loss, the flow spends some time recovering from the packet loss, which we call a *rest*. Then, the next burst begins. Let w be the size of TCP's flow window, measured in segments (for our experiments in §7.3, we would have $w = 4$). Define b as the average number of windows sent in a burst. The goodput of TCP is the number of bytes sent in each burst, which is $w \cdot b \cdot \mathrm{MSS}$, divided by the duration of each burst. A burst lasts for the time to transmit b windows of data, plus the time to recover from the packet loss that ended the burst. The time to transmit b windows is $b \cdot \mathrm{RTT}$. We define t_{rec} to be the time to recover from the packet loss. Then we have

$$B = \frac{w \cdot b \cdot \mathrm{MSS}}{b \cdot \mathrm{RTT} + t_{\mathrm{rec}}}. \tag{3}$$

The value of b depends on the packet loss rate. We define a new variable, p_{win}, which denotes the probability that at least one packet in a window is lost. Then $b = \frac{1}{p_{\mathrm{win}}}$.

To complete the model, we must estimate t_{rec} and p_{win}.

The value of t_{rec} depends on whether the retransmission timer expires (called an RTO) or a fast retransmission is performed. If an RTO occurs, the total time lost is the excess time budgeted to the retransmit timer beyond one RTT, plus the time to retransmit the lost segments. We denote the time budgeted to the retransmit timer as ETO. So the total time lost due to a timeout, assuming it takes about 2 RTTs to recover lost segments, would be $(\mathrm{ETO} - \mathrm{RTT}) + 2 \cdot \mathrm{RTT} = \mathrm{ETO} + \mathrm{RTT}$. After a fast retransmission, TCP enters a "fast recovery" state [17, 66]. Fast recovery requires buffer space to be effective, however. In particular, if the buffer contains only four TCP segments, then the lost packet, and three packets afterward which resulted in duplicate ACKs, account for the entire send buffer; therefore, TCP cannot send new data during fast recovery, and instead stalls for one RTT, until the ACK for the fast retransmission is received. In contrast, choosing a larger send buffer will allow fast recovery to more effectively mask this loss [122].

As discussed in §7.3, these two types of losses may be caused by different factors. Therefore, we do not attempt to distinguish them on basis of probability. Instead, we use a very simple model: $t_{\mathrm{rec}} = \ell \cdot \mathrm{RTT}$. The constant ℓ can be chosen to describe the number of "productive" RTTs lost due to a packet loss. Based on the estimates above, choosing $\ell = 2$ seems reasonable for our experiments in §7 which used a buffer size of four segments.

To model p_{win}, we assume that, in each window, segment losses are independent. This gives us $p_{\mathrm{win}} = 1 - (1-p)^w$, where p is the probability of an individual segment being lost (after link retries). Because p is likely to be small (less than 20%), we apply the approximation that $(1-x)^a \approx 1 - ax$ for small x. This gives us $p_{\mathrm{win}} \approx wp$.

Applying these equations for t_{rec} and p_{win}, along with some minor algebraic manipulation to put our equation in a similar form to Equation 2, we obtain our model for TCP performance in LLNs, for small w and p:

$$B = \frac{\mathrm{MSS}}{\mathrm{RTT}} \cdot \frac{1}{\frac{1}{w} + \ell p} \tag{4}$$

Equation 1, stated in §7.4, takes $\ell = 2$, as discussed above.

Generalizing the model. In §8, we generally use a smaller MSS (3 frames) than we used in §7. Furthermore, duty-cycling increases the RTT. It is natural to ask whether our conclusions in §7, on which the model is based, still hold in this setting. With a sleep interval of 100 ms, we qualitatively observed that, although `cwnd` tends to recover more slowly after loss, due to the smaller MSS and larger RTT, it is still "maxed out" past the BDP most of the time. Therefore, we expect our conclusion, that TCP is more resilient to packet loss, to also apply in this setting.

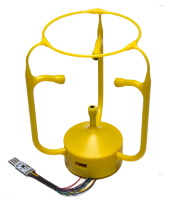
One may consider adapting our model for this setting by choosing a larger value of ℓ to reflect the fact that `cwnd` recovers from loss less quickly due to the smaller MSS. It is possible, however, that one could derive a better model by explicitly modeling the phase when `cwnd` is recovering, similar to other existing TCP models (in contrast to our model above, where we assume that the TCP flow is binary—either transmitting at a full window, or in backoff after loss). We leave exploration of this idea to future work.

D Anemometry: An LLN Application

An *anemometer* is a sensor that measures air velocity. Anemometers may be deployed in a building to diagnose problems with the Heating, Ventilation, and Cooling system (HVAC), and also to collect air flow measurements for improved HVAC control. This requires anemometers in difficult-to-reach locations, such as in air flow ducts, where it is infeasible to run wires. Therefore, anemometers must be battery-powered and must transmit readings wirelessly, making LLNs attractive.

We used anemometers based on the Hamilton platform [19], each consisting of four ultrasonic transceivers arranged as vertices of a tetrahedron (Figure 13). To measure the air velocity, each transceiver, in turn, emits a burst of ultrasound, and the impulse is measured by the other three transceivers. This process results in a total of 12 measurements.

Calculating the air velocity from these measurements is computationally infeasible on the anemometer itself, because Hamilton does not have hardware floating point support and the computations require complex trigonometry. Measurements must be transmitted over the network to a server that


(a) Anemometer

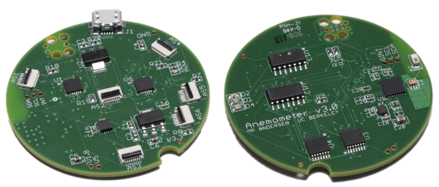

(b) Hamilton-based PCB (bottom and top)

Figure 13: Hamilton-based ultrasonic anemometer

processes the data. Furthermore, a specific property of the analytics is that it requires a contiguous stream of data to maintain calibration (a numerical integration is performed on the measurements). Thus, the application requires a high sample rate (1 Hz), and is sensitive to data loss. A protocol for *reliable* delivery, like TCP or CoAP, is therefore necessary.

We note that the 1 Hz sample rate for this application is much higher than the sample rate of most sensors deployed in buildings. For example, a sensor measuring temperature, humidity, or occupancy in a building typically only generates a single reading every few tens of seconds or every few minutes. Furthermore, each individual reading from the anemometer is quite large (82 bytes), given that it encodes all 12 measurements (plus a small header). Given the higher data rate requirements of the anemometer application, we plan to use a higher-capacity battery than the standard AA batteries used in most motes. The higher cost of such a battery is justified by the higher cost of the anemometer transducers.

Comb Decoding towards Collision-Free WiFi

Shangqing Zhao, Zhe Qu, Zhengping Luo, Zhuo Lu, Yao Liu
University of South Florida, Tampa FL 33620.

Abstract

Packet collisions happen every day in WiFi networks. RTS/CTS is a widely-used approach to reduce the cost of collisions of long data packets as well as combat the hidden terminal problem. In this paper, we present a new design called comb decoding (CombDec) to efficiently resolve RTS collisions without changing the 802.11 standard. We observe that an RTS payload, when treated as a vector in a vector space, exhibits a comb-like distribution; i.e., a limited number of vectors are much more likely to be used than the others due to RTS payload construction and firmware design. This enables us to reformulate RTS collision resolution as a sparse recovery problem. We create algorithms that carefully construct the search range for sparse recovery, making the complexity feasible for system design and implementation. Experimental results show that CombDec boosts the WiFi throughput by 33.6% – 46.2% in various evaluation scenarios.

1 Introduction

CSMA/CA is fundamental for WiFi to coordinate multiple nodes to access the wireless channel. RTS/CTS is a widely-used mechanism in WiFi, which uses short RTS packets for fast collision inference, transmission path check as well as combating the hidden terminal problem [5]. In many early WiFi products, RTS/CTS was disabled due to the concern of overhead [32]. With the substantial increase of data demand in recent years, WiFi data packets become longer and longer, and today's WiFi devices send an RTS packet when the size of a data packet exceeds a given threshold. The threshold is usually set to around 2,300 bytes [1, 4, 6, 7, 9] to balance the performance and the overhead. RTS/CTS is also used in advanced WiFi functionalities, such as beamforming and MU-MIMO [5]. In addition, global RTS/CTS [41, 45, 64] has also been proposed for cross-technology communications.

The essence in the RTS/CTS mechanism is to trade a small cost of the RTS collision for a potentially large cost of data collision. In this paper, we revisit the RTS collision problem and present the comb decoding (CombDec) system to resolve RTS collisions without changing the 802.11 standard and thus improve the wireless channel utilization as well as the network throughput.

The observation behind designing CombDec is that the data payload of an RTS packet, when treated as a vector in a vector space, exhibits a comb-like distribution. Specifically, the RTS content consists of 160 bits, which leads to 2^{160} possibilities. We find that the standard-structured data fields in RTS actually result in at most around 2^{21} possible contents in today's WiFi networks. Therefore, the probability distribution of such contents will exhibit a comb-like shape: only up to 2^{21} out of 2^{160} contents having non-zero probabilities.

As a result, we reformulate the RTS collision problem as a weighted sum problem: we consider the received signal due to an RTS collision is the sum of all 2^{21} RTS contents transmitted at the same time, but with different channel weights. An RTS content has a non-zero channel weight if it is actually transmitted, and zero weight otherwise. Then, resolving the collision is equivalent to solving for the channel weight for each RTS content. The key observation to solve the problem is that a vast majority of the 2^{21} channel weights should be zeros because a collision involves only a few RTS packets in a real-world network. In other words, the vector that includes all channel weights is sparse, which opens a path to use sparse recovery [18, 19, 47] to resolve RTS collisions.

One significantly challenging issue around collision resolution based on sparse recovery is the computational complexity because we start from around 2^{21} possibilities of RTS signals to resolve a collision. To cope with this issue, we analyze how a key RTS data field, `Duration` (that specifies the time duration an RTS packet reserves), is constructed in state-of-the-art 802.11 firmware. A comprehensive set of 802.11ac packet traces are also collected to understand the distribution of `Duration` in various scenarios. It is found that today's firmware imposes extra restrictions on `Duration` and is biased towards a limited number of value selections, and the distribution of `Duration` in real-world packets is highly uneven and patterned. Based on this observation, CombDec is designed with two key components: (α, β)-construction and γ-decimation, which adaptively narrow down the search range in sparse recovery to a set of only hundreds of potential RTS signals, making system design of collision resolution practical for WiFi networks.

We implement CombDec in a 20-node network testbed, and evaluate it in different scenarios. Experimental results demonstrate that our design has both direct and indirect impacts on today's WiFi systems and setups.

Direct Impact: Today's WiFi devices usually adopt a conservative RTS threshold (i.e., around 2300 bytes), which results in 30% - 45% data transmissions initiated by RTS (according to our packet trace collection and analysis). By directly using CombDec with current RTS settings, we find via experiments that CombDec is able to decode 98% of two-RTS collisions and improve the network throughput by up to 23.3%.

Indirect Impact: CombDec offers a new capability of decoding RTS collisions and in fact encourages changing today's WiFi setups for more RTS transmissions. Therefore, by reducing the RTS threshold to zero and letting every device always send RTS before data (indicating that most collisions in the network become RTS collisions), CombDec significantly improves the network throughput by up to 46.6% in experimental evaluations.

The design of CombDec is the first systematic work towards resolving RTS collisions in WiFi networks. It is non-invasive and redefines the role of the RTS functionality and pushes WiFi towards a collision-free environment.

2 Motivation and Design Intuition

In this section, we introduce the motivation and key idea of reformulating the problem of packet collisions.

2.1 Packet Collision and Resolution

We use a noise-free, flat-fading uplink scenario as a simple motivating example: Alice and Bob send their packets to the AP at the same time. Alice's and Bob's packets consist of L time-domain baseband symbols, represented by vectors[1] $\mathbf{x}_A \in \mathcal{X}$ and $\mathbf{x}_B \in \mathcal{X}$, respectively, where $\mathcal{X} \subset \mathbb{C}^{L\times 1}$ denotes the set of all possible baseband symbol vector for $\mathbf{x}_A$ and $\mathbf{x}_B$, and $\mathbb{C}^{L\times 1}$ is the L-dimensional complex vector space.

Then, the received signal at the AP can be written as

$$\mathbf{y} = h_A\mathbf{x}_A + h_B\mathbf{x}_B, \tag{1}$$

where $h_A, h_B \in \mathbb{C}$ ($\mathbb{C}$ denotes the complex plane) are the channel gains from Alice and Bob to the AP, respectively.

If we look at the collision (1) and assume that Alice's and Bob's signals go through similar channel conditions to the AP, Alice's or Bob's signal will have an SNR around 0dB due to mutual interference. Simply given (1), the AP is less likely to recover Alice's or Bob's signal due to two major reasons.

- If the AP adopts a traditional decoding design, it cannot decode a signal with SNR around 0dB, because an acceptable SNR is usually 10dB or above [29] for WiFi.
- Although multi-user detection [59] has been developed as a vital solution to decode multiple user's signals, this technique in general requires that users employ distinct spread spectrum codes [60] to differentiate themselves at the signal level. Nonetheless, there is no such code design in WiFi.

Apparently, additional information is needed to resolve the collision (1). Our key observation is that the RTS packet format itself provides valuable information for collision resolution in a WiFi network.

2.2 Anatomy of RTS in WiFi

The RTS/CTS mechanism lets a sender reserve the channel by sending an RTS packet first. Once the receiver replies with a CTS packet, the sender transmits the data packet. The RTS packet specifies a network allocation vector (NAV), which is the total time duration it wants to reserve, including the time durations of the CTS, the data and the ACK.

The data payload in an RTS packet consists of 20 bytes or 160 bits, and we denote it as an RTS data vector $\mathbf{b} \in [0,1]^{160}$, where $[0,1]^{160}$ is the space for all vectors with length 160, whose element is either 0 or 1. At the PHY layer, the data vector $\mathbf{b}$ is interleaved, coded and modulated into a signal vector $\mathbf{x} \in \mathcal{X}$, where $\mathcal{X}$ is the set of all values of $\mathbf{x}$. These processes together can be denoted as a one-to-one function mapping $f : [0,1]^{160} \to \mathcal{X}$, which converts the RTS data vector $\mathbf{b}$ to the RTS signal vector $\mathbf{x} = f(\mathbf{b})$. As f is one-to-one correspondence, $|\mathcal{X}| = 2^{160}$ ($|\cdot|$ denotes the cardinality, or the number of elements, of a set).

We observe that all RTS data vectors in $[0,1]^{160}$ are not equally probable in the real world because all data fields in RTS are well structured and specified.

- `FrameControl` contains 2 bytes specified in 802.11.
- `Duration` is the 2-byte NAV in microseconds. The last bit is set to 0 and thus it holds up to 2^{15} values.
- `RA` and `TA` (6 bytes each) are the destination and source addresses, respectively. As today's WiFi is widely used for Internet access, stations communicate mostly with the AP. The AP knows that `RA` in an RTS packet sent to it is its own address and `TA` should be the address of one of its stations. Suppose that a dense network can support 2^6 stations (e.g., Linksys EA8500 firmware supports up to 51 stations [11]) and the number of possible values of `TA` in RTS is 2^6.
- `FCS` (4 bytes) is for error detection and relies on other data fields. It provides no additional information.

Thus, from the AP's perspective, the number of RTS data vectors of interest is 2^{15} (from `Duration`) $\times$ 2^6 (from `RA`/`TA`) = 2^{21} in the full RTS vector space $[0,1]^{160}$. As a result, the

[1]Throughout this paper, a vector is by default a column vector instead of a row vector, unless otherwise specified.

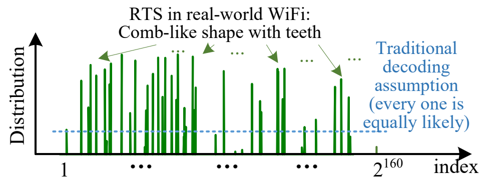


Figure 1: Example: distribution of RTS signal vectors.

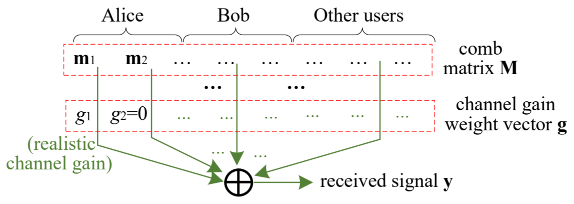


Figure 2: Reformulation of network collision.

number of RTS signal vectors of interest is also 2^{21} in the signal vector space $\mathcal{X}$ with $|\mathcal{X}| = 2^{160}$. If we index all vectors in $\mathcal{X}$ from 1 to 2^{160} and measure via the probability distribution how each vector is likely to be seen in the real world, we will obtain a comb-shaped distribution similar to the example shown in Figure 1. We call an RTS signal vector a *tooth vector* if the probability that it can be seen at the AP is positive. Although the index goes from 1 to 2^{160} in Figure 1, the number of tooth vectors should be no less than 2^{21} according to our analysis. The comb-shaped distribution is in evident contrast to the traditional decoding assumption (also illustrated in Figure 1) that all potential values of a signal vector are equally probable [31]. This opens a door for us to go beyond traditional decoding to resolve RTS collisions.

2.3 Idea of Collision Resolution

Based on the observation from Figure 1, we present the basic idea regarding how to resolve an RTS collision.

2.3.1 Problem Reformulation

Denote by $\mathcal{M} = \{\mathbf{m}_i\}_{i\in[1,M]}$ ($M = |\mathcal{M}|$) the set of tooth vectors. Define the comb matrix $\mathbf{M} = [\mathbf{m}_1, \mathbf{m}_2, \cdots, \mathbf{m}_M]$ (i.e., each column in $\mathbf{M}$ is a tooth vector). The AP can preconstruct the comb matrix $\mathbf{M}$ by inserting all possible RTS to $\mathbf{M}$ to form the columns. The AP's goal is to find exactly which ones in $\mathbf{M}$ are actually involved in the collision.

Mathematically, we reformulate the collision problem to an equivalent one: assume that all tooth vectors in $\mathbf{M}$ are transmitted to the AP, but they go through different wireless channels. In particular, the tooth vectors involved in the actual collision go through the wireless channels with realistic channel gains, but those not involved in the collision go through the channels with zero channel gains. For example, in Figure 2, Alice, Bob and other users have different tooth vectors. The received signal $\mathbf{y}$ is considered as the sum of all these tooth vectors weighted by different channel gains. The channel gain weight of a tooth vector is the realistic channel gain if it is indeed transmitted, and zero otherwise. If Alice's transmitted signal is $\mathbf{m}_1$, the channel gain weight g_1 for $\mathbf{m}_1$ is the real channel gain between Alice and the AP, and the weights for the rest of Alice's tooth vectors are all zeros (e.g., $g_2 = 0$ as Alice transmits $\mathbf{m}_1$ not $\mathbf{m}_2$). As such, the received signal $\mathbf{y}$ can be reformulated as

$$\mathbf{y} = \sum_{i=1}^{M} \mathbf{m}_i g_i = \underbrace{[\mathbf{m}_1, \mathbf{m}_2, \cdots, \mathbf{m}_M]}_{\text{comb matrix } \mathbf{M}} \ \underbrace{[g_1, g_2, \cdots, g_M]}_{\text{channel gain weight vector } \mathbf{g}}{}^{T}, \tag{2}$$

where $\mathbf{g}$ is called the channel gain weight vector and $\cdot^T$ denotes the matrix transpose. Based on (2), collision resolution is equivalent to solving for unknown $\mathbf{g}$ given $\mathbf{y}$ and $\mathbf{M}$. Then, the tooth vectors actually involved in the collision correspond to non-zero elements in the solved $\mathbf{g}$.

2.3.2 Solution based on Reformulation

There are two key observations on the reformulation in (2).

- The unknown channel gain weight vector $\mathbf{g}$ is sparse in a real-world network because of two reasons: (i) There is no self-collision. As shown in Figure 2, if Alice sends a tooth vector $\mathbf{m}_1$, we have $g_1 \neq 0$; and any other tooth vector belonging to Alice will have a zero channel gain weight (e.g., $g_2 = 0$) since there is no way Alice transmits both $\mathbf{m}_1$ and $\mathbf{m}_2$. (ii) Because of the random backoff in WiFi, a collision is likely caused by only several users transmitting at the same time. Thus, $\mathbf{g}$ should include only several non-zero elements.
- The comb matrix $\mathbf{M}$ should exhibit a nearly random matrix property. Each tooth vector $\mathbf{m}_i$ in $\mathbf{M}$ is mapped from an RTS data vector through interleaving and error-correction coding. Their main purpose is to scramble and re-map all bits into a larger bit space in a (nearly) random way such that the error-correction performance approaches the random coding performance in Shannon's capacity [24].

Given the sparse property and nearly random matrix property, the channel gain weight vector $\mathbf{g}$ should be recovered with high probability by $\mathcal{L}_1$-norm minimization according to the theory of compressive sensing [26, 28, 38]. Thus, resolving an RTS collision leads to the following optimization.

$$\begin{aligned} \text{Given:}\quad & \text{comb matrix } \mathbf{M} \text{ and received signal } \mathbf{y}, \\ \text{Objective:}\quad & \mathbf{g}_{\text{solution}} = \arg\min \|\mathbf{g}\|_1, \text{ subject to } \mathbf{y} = \mathbf{M}\mathbf{g}, \end{aligned} \tag{3}$$

where $\|\mathbf{g}\|_1$ is the $\mathcal{L}_1$-norm of $\mathbf{g}$ (i.e., $\|\mathbf{g}\|_1 = \sum_{g_i \in \mathbf{g}} |g_i|$).

The theoretical framework lays out a promising path towards resolving RTS collisions. Although the $\mathcal{L}_1$-norm minimization in (3) can be solved by many efficient algorithms

[17–19, 27, 62], directly applying (3) to collision resolution incurs an unbearable cost because the comb matrix **M** consists of up to $2^{21} = 2,097,152$ tooth vectors according to our initial analysis in Section 2.2. Thus, significant challenges exist to make collision resolution meaningful and practical.

3 Construction of Comb Matrix

The initial step towards our CombDec design is to find a way to reduce the size of the comb matrix **M** (that can include 2^{21} tooth vectors) for a low-cost solution. In this section, we analyze the 802.11 standard, firmware and packet traces to show that there is a feasible way to significantly reduce the size of 2^{21}. Then, we design two algorithms, (α, β)-construction and γ-decimation, to reduce 2^{21} to only a few hundreds, while maintaining the high performance for collision resolution. The use of (α, β)-construction and γ-decimation clears the major hurdle towards system implementation.

3.1 Standard and Firmware based Analysis

As aforementioned in Section 2.2, **M** consists of 2^{21} tooth vectors because of `Duration` and `RA/TA` fields in RTS. The `Duration` field specifies the 15-bit NAV in microseconds with $2^{15} = 32,768$ potential values, which largely contribute to the size of **M**. 802.11 specifies that the NAV is computed as one data packet duration, plus one CTS, one ACK, and three SIFS durations. Given a network setup, SIFS, CTS and ACK durations are usually fixed (e.g., SIFS is fixed to be 16 μs in 802.11ac at 5GHz). Thus, the value space of the NAV depends dominantly on the value space of the time duration of a data packet. In 802.11, the time duration of a data packet is bounded by aPPDUMaxTime, the maximum time duration of a data packet (in μs), and indirectly bounded by aPSDUMaxLength, the maximum payload length of a data packet (in bytes). As aPPDUMaxTime for 802.11ac is $5,484\mu s$, the space size of the NAV is reduced from 32,768 to at most 5,484 in the 802.11ac network.

Today's WiFi chipsets may still avoid transmitting a long packet (close to $5,484\mu s$) due to the cost consideration or hardware limitations. Hence, drivers implement their own packet length constraints on a data packet, which is usually less than the standard-defined aPSDUMaxLength or aPPDUMaxTime. We perform comprehensive code analysis on WiFi drivers and find that different vendors indeed pose different constraints on their own chipset, further limiting the value selection of the NAV. The detailed firmware analysis can be found in Appendix A.

As a result, we can leverage these constraints to further reduce the value space of the NAV in RTS. However, many WiFi drivers (in particular 802.11ac ones) are still proprietary and distributed in the binary form. It is not practical to study every WiFi chipset/firmware and optimally minimize the value space of the NAV in RTS packets. In what follows, we analyze real-world packet traces to develop a generic way to narrow down the value space.

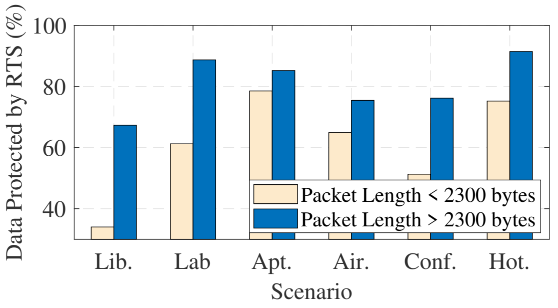


Figure 3: Percentages of data packets protected by RTS.

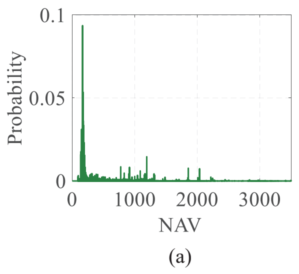
3.2 Packet Trace based Analysis

The key to reducing the size of the comb matrix is through shrinking the value space of NAV in RTS. We have shown that a WiFi driver can restrain the value space of NAV. Moreover, implementation-dependent rate control and data aggregation in proprietary WiFi drivers are not likely to produce uniformly distributed NAV values, but may be more biased towards certain selections and yield a NAV distribution similar to Figure 1. Our objective is to collect massive packet traces to understand the NAV distribution. Then, we create generic algorithms to select those NAVs that are the most likely to be seen for constructing the comb matrix **M**.

The first step towards understanding the NAV distribution in real-world RTS packets is to collect a substantially large number of packet traces for analysis. As no set of 802.11ac packet data is publicly available, we conducted our own measurements and collected in total 1.3 TB packet trace data with 2.33 billion packets in realistic environments[2], including (i) a public library (65.21 GB), (ii) three academic conferences (31.14 GB), (iii) five residential communities (65.69 GB), (iv) three major-brand hotels (109.48 GB), (v) four major US airports (88.5 GB), (vi) a university research lab (938.81 GB). The library, conference, and airport data traces were measured only within the business hours (i.e., 9am–5pm).

Figure 3 shows the the percentages of data packets that are protected by RTS among all data packets collected in different scenarios. Although a typical RTS threshold is set to around 2300 bytes [1, 4, 6, 7, 9], we see from Figure 3 that data packets of less than 2300 bytes are still likely to be protected by RTS. For example, in the hotel scenario, 78.55% data packets are initiated by RTS even when their lengths are less than 2300 bytes. Moreover, around 70% - 90% data packets of over 2300 bytes are protected by RTS in different scenarios. Overall, we observe from Figure 3 that RTS is still

[2]Note that the payloads of all data packet were removed after the collection to avoid the privacy concern.



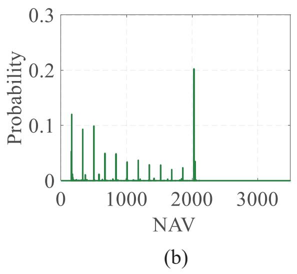
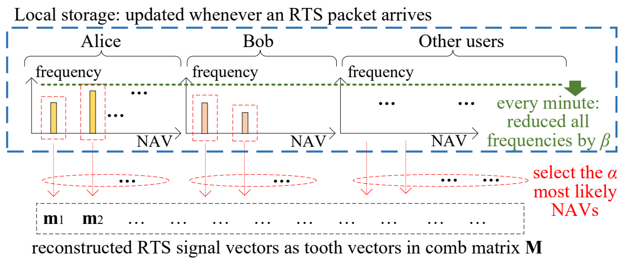


Figure 4: Distribution of NAVs from (a) the airport dataset, and (b) all Belkin devices in all datasets.



Local storage: updated whenever an RTS packet arrives
Alice
Bob
Other users
frequency
frequency
frequency
NAV
NAV
NAV
every minute: reduced all frequencies by β
select the α most likely NAVs
$\mathbf{m}_1$ $\mathbf{m}_2$
reconstructed RTS signal vectors as tooth vectors in comb matrix **M**

Figure 5: (α, β) construction running at the AP.

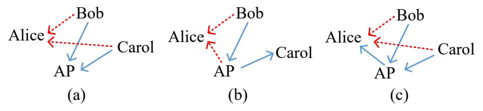


Figure 6: Different collision scenarios in Alice's view.

intensively used in today's Wi-Fi networks.

Looking into the NAV values in RTS packets, we observe that these values are unevenly distributed. For example, Figure 4(a) shows the NAV distribution of RTS packets in the airport dataset, which reveals that many NAV values (particularly around 230 μs) are much more likely to be observed than the others. The NAV distribution also depends on a WiFi driver. For example, Figure 4(b) plots the distribution of the Belkin WiFi driver measured from RTS packets sent by Belkin devices (recognized by MAC addresses) in all datasets. The figure shows that the distribution is quite patterned, indicating that the driver has several NAV levels to construct payloads. We observe uneven or patterned distributions in all datasets and offer a more detailed analysis in Appendix B.

Thus, if we only choose the most likely NAV values (instead of all possible values) to construct the comb matrix M, the size of M should be substantially reduced. In addition, our design should not be device/firmware specific. For example, it may be possible to select NAV values based on the pattern of Belkin devices in Figure 4(b). But this method is too cumbersome because we have to examine the behaviors of all different WiFi devices. Our strategy is to use an online algorithm that actively computes the NAV distribution of a device, and then selects the most likely NAV values from the computed distribution to construct the comb matrix **M**.

3.3 The (α, β)-Construction Algorithm

To select the most likely NAV values to construct **M**, a node (either the AP or a station) should store the distribution of the NAV values in RTS packets from every other node in a network. In addition, the node should keep updating the storage to account for nodes joining or leaving the network. To this end, we propose the (α, β)-construction algorithm running at individual nodes to construct **M**.

3.3.1 Algorithm Design

For a node that runs the algorithm, it records the frequency (i.e. the number of appearances) of a NAV value in RTS packets transmitted by every other node in the network. When a new RTS packet with a NAV value is decoded, the node will increase the frequency of that NAV value by one in its local storage. There are two key factors α and β in the algorithm.

- Coverage factor α: For every other node in the network, the algorithm selects the α NAV values with the highest frequencies to form the tooth vectors and the comb matrix.
- Forgetting factor β: The algorithm decreases the frequencies of all NAV values by β every minute. The minimum frequency is always set to be zero. And if the frequencies of all NAV values associated with a node become zero, the node's information will be all removed from the storage as it is considered no longer active or out of the network.

The (α, β)-construction algorithm works differently for the AP and stations. Figure 5 shows how it works at the AP: the AP stores the frequency of each possible NAV value from each station. When a collision happens at the AP, it knows the collision must be due to at least two stations (which could be Alice, Bob, or others) transmitting to it. Thus, the AP selects the α most likely NAV values from Alice, constructs an RTS data vector using each of these values, together with Alice's MAC address as `TA` and its own MAC address as `RA`, then maps each RTS data vector by function f (including interleaving, error-correction coding, modulation, IFFT) in Section 2.2 to a signal vector (which is a tooth vector in the comb matrix **M**). Then, the AP repeats the same process for Bob and all other stations to finally obtain the full **M**.

A station's construction of the comb matrix differs from the AP. For example, as shown in Figure 6, Alice observes a collision (a) when two other stations Bob and Carol are transmitting to the AP, (b) when one other station Bob is transmitting to the AP and at the same time the AP is transmitting to a third station Carol, or (c) when the AP is transmitting to Alice while Bob and Carol are transmitting to the AP. In all three cases, collision resolution is only meaningful for Alice in case (c) because the collision in case (c) includes the AP's

Table 1: Miss rate and average size of **M**.

α=600 β=10	Miss rate	Ave. # of nodes	# of tooth vectors in **M**
Library	6.9 %	12.8	7680
Conferences	3.6 %	11.6	6960
Apartments	1.5 %	6.5	3900
Hotels	0.7 %	10.3	6180
Airports	8.8 %	18.8	11280
Lab	5.7 %	6.3	3780

signal intended for Alice. Even when Alice successfully resolves cases (a) and (b), Alice only knows that there is no signal of interest and then stops. Therefore, the construction is sufficient for Alice as long as case (c) can be resolved by Alice. According to case (c), a station should construct the comb matrix by using RTS signal vectors from the AP to itself and other RTS signal vectors from other stations to the AP. The construction process is similar to Figure 5.

3.3.2 Selections of (α,β) and Cost Evaluations

The size of the resultant comb matrix **M** depends on both α and β. In particular, α is the number of tooth vectors from one node, and β in fact determines how many nodes will be used in constructing **M** because (i) all frequencies are decreased by β every minute and (ii) a node will be removed from the construction when all its frequencies become zero. The algorithm with a larger β forgets nodes faster, thereby reducing the number of nodes used for constructing **M**.

The performance of (α,β)-construction can be evaluated by the miss rate, defined as the probability that when an RTS packet arrives at a node, **M** constructed by the algorithm at the node does not include the NAV value in the RTS packet. A large α and a small β are able to reduce the miss rate, but at the same time increase the size of **M**, incurring more cost.

Our objective is to find the pair of (α,β) to balance the miss rate and the complexity for general WiFi scenarios. To this end, we simulate a WiFi network in each of the packet datasets, replay all collected packets to simulate RTS arrivals at each node, and measure the miss rate of the algorithm with different values of α and β. Table 1 shows one selection of (α,β)=$(600,10)$ for all scenarios that achieves a good balance between the miss rate and the size of **M** (measured by α multiplying the average number of nodes used for constructing **M**). We can see that all miss rates are below 9% with around 4,000–12,000 tooth vectors in **M**.

3.4 The γ-Decimation Algorithm

Through 802.11 standard analysis, firmware analysis, packet trace analysis and (α,β)-construction, we have dramatically shrink the size of **M** from the initial 2,097,152 tooth vectors to 12,000 or fewer vectors. All these push the optimization in (3) to the practice. However, finding the $\mathcal{L}_1$-norm minimization with $12,000$ vectors in (3) still incurs a substantial cost. We propose γ-decimation to further reduce such a cost while maintaining the high performance.

Denote by M the number of tooth vectors in **M** constructed by (α,β)-construction. The basic idea of the γ-decimation algorithm ($\gamma > 1$ is called decimation rate) is to select, based on the received signal vector **y**, M/γ vectors out of all M tooth vectors in **M** to form a decimated comb matrix $\mathbf{M}'$.

The design intuition is that the received signal **y** contains only several tooth vectors in **M** that we aim to find out. If we compute the correlation between **y** and each tooth vector $\mathbf{m}_i$ in **M**, defined as $C(\mathbf{y},\mathbf{m}_i) = \|\mathbf{m}_i^H\mathbf{y}\|_2$ ($\cdot^H$ denotes conjugate transpose and $\|\cdot\|_2$ denotes the $\mathcal{L}_2$-norm), we then obtain M correlation values. Due to the property of correlation, we should observe a high correlation value if a tooth vector is indeed included in **y**, and a low correlation value otherwise. However, due to channel noise and limited length of tooth vectors, some tooth vectors not in **y** may also exhibit high correlation values. But it is not necessary to exactly identify which tooth vectors with high correlation values are indeed in **y** at this stage, γ-decimation just chooses M/γ tooth vectors that have the highest correlation values to form the decimated comb matrix $\mathbf{M}'$. It is very likely that tooth vectors involved in the collision are included in $\mathbf{M}'$ as long as M/γ is sufficiently large. We provide theoretical analysis for the performance of γ-decimation and show that all RTS signals involving a collision will survive the decimation and be included in $\mathbf{M}'$ with high probability in Appendix C.

As a result, the final comb matrix for (3) is constructed as follows: (i) (α,β)-construction constructs the comb matrix **M** with M tooth vectors (this steps ensures a low miss rate), (ii) γ-decimation decimates **M** into the decimated comb matrix $\mathbf{M}'$ with only M/γ tooth vectors (this steps ensures that tooth vectors involving the actual collision are preserved with high probability), and (iii) the decimated comb matrix $\mathbf{M}'$ is used in (3) for collision resolution to finally identify which tooth vectors are included in the collided signal **y**.

To make (3) feasible for today's systems, $\mathbf{M}'$ should have only hundreds of tooth vectors. As (α,β)-construction maintains up to around $12,000$ vectors (shown in Table 1), the decimation rate γ should be around 12 or more.

3.5 Complexity Analysis

In the following, we estimate the computational complexity and storage complexity of CombDec.

3.5.1 Computational Complexity

We evaluate CombDec's complexity by comparing it with a benchmark, which is the complexity to decode a typical 802.11 data packet with 40MHz bandwidth, 3/4 convolu-

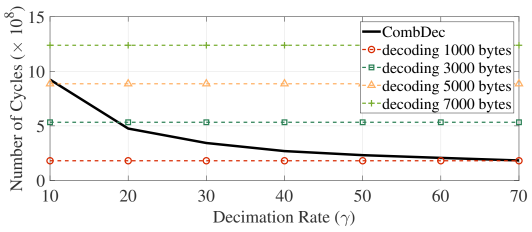


Figure 7: Number of cycles: CombDec vs benchmark.

tional coding, and 64QAM. We describe the major computational operations involved in the benchmark and CombDec.

• The benchmark complexity is proportional to the data payload length, denoted by N bytes. Decoding a data packet requires the FFT and Viterbi algorithms. The packet contains $0.0247N$ OFDM symbols plus 4 PHY headers symbols. The FFT on each symbol requires $64\log(128)$ complex multiplications and $128\log(128)$ complex additions [10]. In addition, the Viterbi algorithm incurs $4^K(10.67N+252)$ real additions and $2^K(10.67N+252)$ real comparisons, where K is the constraint length of convolutional code [12].

• In CombDec, suppose (α,β)-construction maintains M tooth vectors of length L, the correlation of the received signal with each of the tooth vector in γ-decimation needs a total of ML complex additions and multiplications. Among all correlation values, selecting the M/γ largest values incurs $M+M\log(M)/\gamma$ comparisons by using the max heap tree approach [3]. The complexity of $\mathcal{L}_1$ minimization depends on an iterative algorithm and is bounded by the cubic polynomial complexity [62]. In order to estimate an exact computational cost, we use simulations to run the primal-dual interior-point algorithm [47] to solve the $\mathcal{L}_1$-norm minimization in M/γ tooth vectors and compute the average numbers of additions, multiplications and comparisons.

As the complexity involves different types of operations, including additions, multiplications and comparisons. We need to have a unified cost utility metric to measure the overall costs of CombDec and the benchmark. We use the number of general CPU cycles as the utility metric [2]. In particular, one real addition or comparison is counted as one cycle and one multiplication is 4 cycles [2]; and a complex operation is 4 times the number of cycles incurred by its real counterpart [10]. Note that an algorithm or a computational operation can be specifically optimized on a particular signal processing software or hardware platform. Our estimation using CPU cycles is not intended to be exactly accurate for an implementation platform, but serves as an approximate way to demonstrate what computational complexity level CombDec is at when compared with the benchmark.

Figure 7 shows the total numbers of cycles incurred by CombDec and the benchmark. In Figure 7, we let (α,β)-construction form $M = 12,000$ tooth vectors to accommodate the airport scenario in Table 1 and set a typical constraint length $K = 7$ in the Viterbi Algorithm. It is observed from Figure 7 that choosing γ to be in [20, 50] leads to the complexity of CombDec roughly equivalent to decoding a packet with 1000–3000 bytes, which makes CombDec ready for system implementation.

3.5.2 Storage Complexity

CombDec also incurs a storage cost. First, CombDec for a device must store the frequency of each NAV value for any other active device in the network. The cost of storing all NAV frequencies is equal to the NAV space size multiplying the average number of active devices. There are 5,484 possible NAV values in 802.11ac and around 18.8 active nodes under (α,β) construction for the airport scenario (shown in Table 1). Hence, the storage cost is $5484 \times 18.8 \times 1 = 101$ KB when the frequency value is a one-byte integer. Second, after γ-decimation, all tooth vectors should be stored for $\mathcal{L}_1$-minimization. The storage cost is the number of tooth vectors multiplying the length of a tooth vector. When $\gamma \in [20,50]$, the number of decimated tooth vectors in the airport scenario is 240 to 600. If a typical RTS packet is transmitted at 12 Mbps (4 64-subcarrier OFDM symbols) and the element in tooth vector is a 4-byte complex number, the length of a tooth vector is $64\times4\times4 = 1$ KB. Thus, the cost of storing all these tooth vectors is in the range of $[240,600]$ KB.

Overall, the major storage cost is around [341, 701] KB. This cost is also reasonable for today's WiFi systems. For example, Qualcomm's 802.11ac chipset IPQ4018 has an on-chip memory of 256 MB [50] and related APs cost as low as tens of dollars [8, 13].

4 CombDec System Design

The (α,β)-construction and γ-decimation algorithms pave the way for a feasible system solution to (3). In this section, we present CombDec system design. We first introduce the system architecture, then describe each key component.

We design CombDec as an independent decoder in addition to the traditional 802.11 decoder. Figure 8 shows four major components in CombDec: the prologue module, (α,β)-construction, γ-decimation, collision resolution, and the epilogue module. As shown in Figure 8, CombDec is triggered only when decoding of either the PHY header or the PHY payload fails. In the following, we present the designs of individual CombDec Components.

The Prologue Module: The prologue module does all preprocessing before collision resolution.

Combining Multi-path Signal Components: The received signal may include a number of multi-path components with different time offsets. To improve the performance of CombDec under multi-path fading, we use the maximum ratio combining (MRC) [31] to combine the multi-path signal components. Specifically, denote by $s(n)$ the n-th time-domain

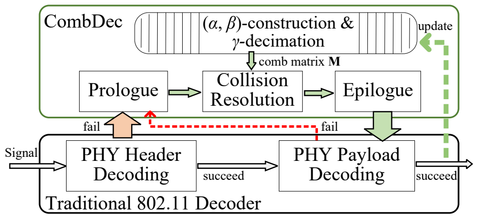


Figure 8: Architecture of CombDec decoder.

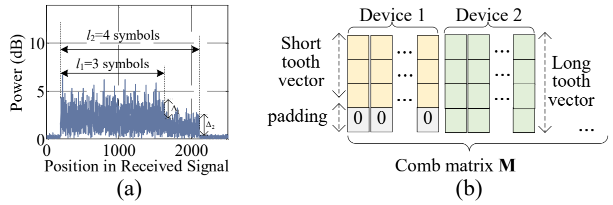


Figure 9: (a) Power level changes in the received signal. (b) Padding in comb matrix construction.

symbol in the received signal. Based on the 802.11 packet preamble, we use a matched filer [32, 58] to detect the time offset and estimate the channel gain of each signal component. Given K components found, we use the MRC [31] to coherently sum all K time-shifted copies of $s(n)$ and obtain $s'(n) = \sum_{k=1}^{K} h_k^* s(n+\delta_k)$, where k-th component having time offset δ_k and channel gain h_k, and h_k^* is the complex conjugate of h_k. Note that signal components may be from different senders in a collision, CombDec does not differentiate them in the prologue module.

Collision Recognition and Early Stop: Next, CombDec decides if $s'(n)$ can be potentially resolved. There are three types of collisions: RTS-only, RTS-data (involving at least one RTS packet and one data packet), and data-only collisions. CombDec will stop if the collision belongs to data-only collision. Note that the recognition does not need to be 100% accurate, it simply provides a way to exclude obvious data-only collisions that cannot be resolved by CombDec.

In WiFi networks, the beginnings of collided packet transmissions are roughly aligned because of CSMA/CA (if we do not consider the hidden terminal problem). Thus, we measure the time durations of different power levels of the received signal to identify the type of a collision. According to 802.11, the data rate for RTS is selected by a station from a limited set of basic rates defined by the AP (e.g., 12 Mbps and 24 Mbps are widely observed in our packet traces). Thus, a RTS time duration can be measured by a very limited number of OFDM symbol durations, e.g., 3 (or 4) OFDM symbol durations for 24 (or 12) Mbps. We record the time duration l_i, from the beginning of the signal $s'(n)$, to the position in $s'(n)$ where the i-th signal power change happens and is larger than a threshold Δ. When the power level reaches the noise floor, we stop and obtain a set of time durations $\mathcal{L} = \{l_i\}_{i\in[1,|\mathcal{L}|]}$. We consider a collision as (i) RTS-only if each value in $\mathcal{L}$ is close to one of the RTS durations corresponding to the basic rate set defined by the AP, (ii) RTS-data if one value is close to an RTS duration and any other value is not close to any of the RTS durations, and (iii) data-only otherwise. Figure 9(a) shows an example of RTS-only collision, where two measured time durations occupy 3 and 4 OFDM symbol durations, respectively, which is recognized as the collision of two RTS packets with different rates.

The Collision Resolution Module: The prologue module outputs either RTS-only or RTS-data signals for collision resolution, (α, β) construction will construct every tooth vectors for the comb matrix. As devices in a network may use different basic rates to transmit RTS packets (leading to different lengths of tooth vectors), CombDec zero-pads the shorter tooth vectors with higher data rates to form the complete comb matrix $\mathbf{M}$ used for collision resolution. Figure 9(b) shows an example of constructing the comb matrix $\mathbf{M}$ by zero-padding shorter tooth vectors. Then, $\mathbf{M}$ is γ-decimated to $\mathbf{M}'$, which is used in the primal-dual algorithm [47] that solves the $\mathcal{L}_1$-minimization in (3). Note that if a collision is RTS-data, the module resolves the collided RTS signal vectors by treating any data signal as the noise, and then leaves the potential decoding of data for the epilogue module.

The Epilogue Module: The collision resolution module yields a small set of potential RTS signal vectors involved in the collision, denoted by $\mathcal{R}$. There are still important questions left: (i) Which RTS in $\mathcal{R}$ a receiver should choose to reply with CTS? (ii) Can we decode the data in the presence of an RTS-data collision? (iii) Moreover, we also need an error detection mechanism to ensure the collision is correctly resolved because errors may happen in CombDec. We first describe how CombDec chooses data or RTS. The AP and stations have different decision making processes.

Decision Making at AP: The AP observes a collision when multiple stations transmit to the AP at the same time.

- Choosing the RTS with the largest NAV: if the collision is RTS-only, the AP chooses the RTS of the largest NAV to reply with CTS. Note that we intend to maximize the channel utilization in this way. A more advanced policy (e.g., considering the utilization and fairness) can be designed and adopted going beyond the main scope of this paper.
- Choosing data over RTS: If the collision is RTS-data, the AP first decodes all RTS packets in the collision resolution module, then proceeds to decode the data. If the data is successfully decoded, the AP chooses data over RTS and sends back the ACK to the sender of the data. This is because data packets are usually longer than RTS packets. Giving priority to data should improve the channel utilization.

Decision Making at Stations: A station observes a collision when multiple other nodes (either other stations or the AP) transmit at the same time. As shown in Figure 6(c), a station only cares about the AP's signal transmitted to it.

• If $\mathcal{R}$ includes an RTS vector from the AP to the station, the collision is due to the AP transmitting to the station and at the same time at least one other station transmitting to the AP, the station always sends CTS to the AP.

• Otherwise, there is no RTS in $\mathcal{R}$ intended for the station. If the collision is RTS-data, the station proceeds to decode the data because the data may be intended for it; otherwise, the station stops.

Error Checking and Data Decoding: Once a decision is made (choosing RTS or data), we must ensure there is no error with the chosen RTS or we must proceed to decode the data. How to proceed with error checking and data decoding? We find that a traditional 802.11 receiver, as shown in Figure 8, already has a PHY payload decoder with the cyclic redundancy check (CRC) mechanism. Thus, we should leverage the existing architecture to perform the decoding and error checking to minimize the complexity of CombDec.

As a result, if CombDec decides to act on a particular RTS signal or to decode a data signal, it uses interference cancelation to remove all other signals from the collided signal. Specifically, if a decision is to decode the data, CombDec removes all RTS signal vectors in $\mathcal{R}$ from the received signal $s'(n)$ and write the resultant signal as $s'_c(n) = s'_c(n) - \sum_{i=1}^{|\mathcal{R}|} r_i(n)g_i$, where $r_i(n)$ and g_i are the n-th element and the channel gain weight of the i-th RTS tooth vector in $\mathcal{R}$, respectively. Similarly, if the decision is to act on the j-th RTS vector (i.e., choose the j-th RTS in $\mathcal{R}$ to reply with CTS), CombDec removes all other RTS tooth vectors from the $s'(n)$ and the resultant signal becomes $s'_c(n) = s'_c(n) - \sum_{i=1,i\neq j}^{|\mathcal{R}|} r_i(n)g_i$.

Finally, the signal $s'_c(n)$ goes from CombDec into the traditional PHY payload decoder, which performs decoding and error checking, then passes the correct RTS or data to the MAC layer for protocol processing (e.g., replying with CTS). In addition, the MAC address and data rate information of a correct RTS packet is stored and its NAV value is also updated in (α, β)-construction as shown in Figure 8.

5 Evaluation

In this section, we evaluate the performance of CombDec. We first introduce the experimental setups, then measure the performance benefits CombDec brings to WiFi networks.

5.1 Setups

Testbed Implementation: We have implemented the prototype of CombDec on 20 USRP X310/300 devices. Each device is equipped with two UBX-160 daughterboards and two VERT 2450 antennas. We implement a basic 802.11ac PHY-MAC architecture with 20-MHz settings: 64 OFDM subcarriers (including 48 data subcarriers), BPSK, QPSK, 16QAM, and 64QAM modulations, Alamouti code based MIMO, and convolutional coding at the PHY layer, and CSMA/CA scheme with an initial contention window size of 8 [5] at the MAC layer. Control packets including RTS, CTS, and ACK are also implemented.

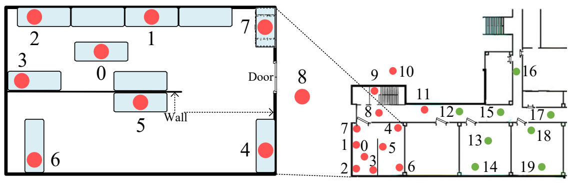


Figure 10: Environment for experiments.

Experimental Settings: We aim to measure the performance of CombDec in a realistic indoor environment inside a campus building shown in Figure 10. Network nodes are placed at various locations and transmit packets whose contents are generated according to our collected 802.11ac packet traces.

Note that we do not implement the 256QAM modulation as we found no single packet using a data rate associated with 256QAM in all collected packet traces. This does not severely affect the performance evaluation since 256QAM is intended only for very high SNR conditions.

We use the following default settings for experiments (unless otherwise specified): (i) all nodes are saturated; i.e., they always have packets to transmit; (ii) α=600, β=10, and γ=20 for CombDec; (iii) the airport dataset is used to generate packets as it represents the most crowded condition in all datasets; (iv) all nodes have the same transmit power.

Evaluation Metrics: We use the following metrics to evaluate the real-time performance of CombDec.

• Success probability of collision resolution is defined as the probability that CombDec recovers exactly all collided RTS signals. The recovery will be considered as a failure if CombDec recovers only a subset of collided RTS signals or mis-identifies an RTS signal not involved in the collision.

• Normalized throughput (or utilization efficiency), is defined as the percentage of the time duration on the wireless channel that is used to deliver data packets. An ideal network should have a normalized throughput of 1. However, control signals (e.g., RTS/CTS) and collisions deteriorate the throughput. We aim to show how much channel utilization efficiency CombDec can improve via resolving RTS collisions.

• Throughput gain ratio, defined as the ratio between the increased normalized throughput from traditional 802.11 decoding to CombDec and the normalized throughput under traditional decoding. Throughput gain ratio can directly reflect how CombDec improves the network performance.

5.2 Success Probability

We first evaluate the success probability of CombDec for collision resolution. In this evaluation, multiple nodes only

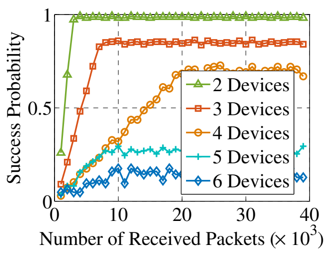


Figure 11: Success probabilities under 2-6 transmitters.

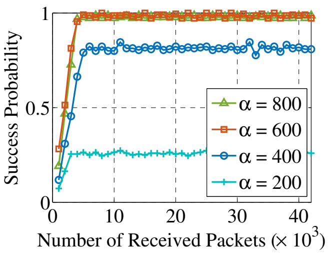


Figure 12: Impact of value of α in (α, β)-construction.

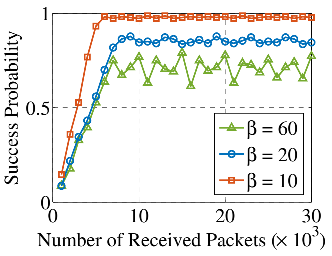


Figure 13: Impact of value of β in (α, β)-construction.

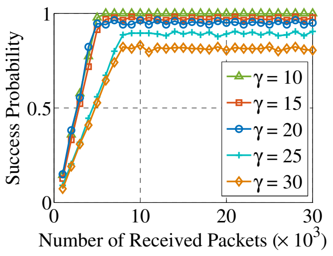


Figure 14: Impact of value of γ in γ-decimation.

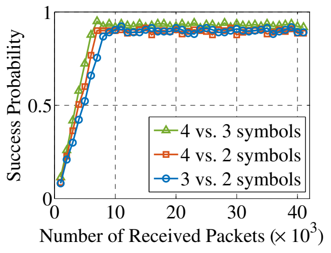


Figure 15: Resolving collisions with different rates.

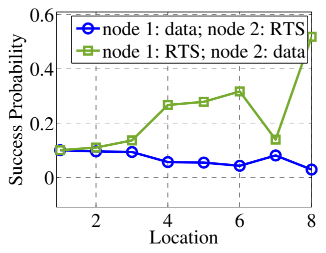


Figure 16: Resolving RTS-data collisions.

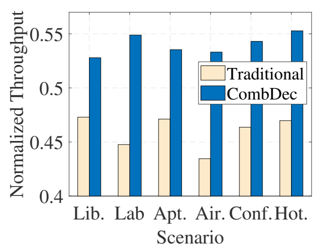


Figure 17: Throughputs under different scenarios.

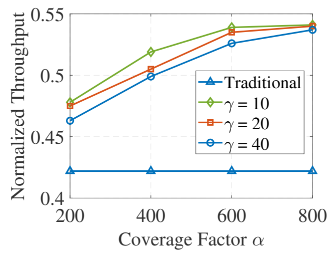


Figure 18: Throughputs with different α and γ.

transmit RTS packets to the AP placed at location 0 in Figure 10, where the success probability is measured.

Impact of Number of Transmissions: We evaluate CombDec's ability to resolve collisions due to two and more transmissions. To this end, we place multiple transmitters at location 1, which send RTS packets at the same time to the AP at location 0. Figure 11 shows the success probabilities that the AP resolves the collision under 2–6 transmitters. The success probability is computed for every 1,000 packets received. It is noted from Figure 11 that as the total number of received packets at the AP increases, CombDec gradually gains the NAV information in RTS packets, and thus the success probability also increases and finally remains stable. It is also observed that CombDec is able to resolve 98% of two-node collisions and 86% of three-node collisions. The performance degrades when the number of transmitters increases. However, a collision caused by 5 or more WiFi nodes is much less frequent because of the random backoff in CSMA/CA.

Values of α and β: We evaluate the impacts of α and β on the success probability. We place two nodes at location 1 that transmit RTS packets at the same time to the AP at location 0. Figure 12 shows that as α goes from 200 to 600, the success probability increases from 0.28 to 0.98; and further increase of α will not substantially improve the success probability. Figure 13 shows the impact of β on the success probability. We observe that when β increases from 10 to 60 (i.e., CombDec forgets the history faster), the success probability reduces from 0.98 to 0.77. From both figures, we can see that the uniform selection of $\alpha = 600$ and $\beta = 10$ yield very high performance for the airport scenario, and accordingly are also suitable for other less crowded scenarios.

Value of γ: We adopt the same setup in Figure 13 to evaluate the impact of γ. Figure 14 illustrates that gradually increasing γ does not severely decrease the success probabilities. For example, when γ becomes 30, the success probability reduces to 0.831. This indicates that adjusting γ can smoothly balance the performance and the implementation cost.

Impact of Zero-Padding: In each of our packet datasets, we find that a majority of RTS packets are transmitted at the same data rate; however, RTS packets with different rates do exist. These packets have different lengths of 2, 3, or 4 OFDM symbols. Therefore, a minority of collisions involving RTS packets with different rates. CombDec uses zero-padding to solve this issue as discussed in Section 4. We measure the ability of CombDec to resolve such a type of collisions. Hence, we place two nodes at location 1 sending RTS packets with different lengths to the AP. Figure 15 shows that the success probabilities remain approximately the same when RTS packets have different lengths in a collision. From Figure 15, we conclude that CombDec has no difficulty in resolving this type of collisions.

RTS-Data Collisions: CombDec also attempts to resolve an RTS-data collision via canceling the RTS signal from the received signal and then performing decoding. To evaluate such an ability, we place one node (node 1) at location 1 to send RTS to the AP, and place another node (node 2) at one

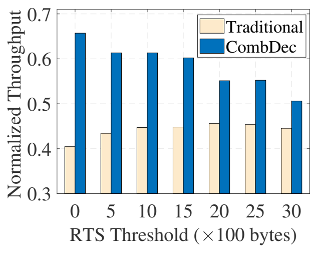


Figure 19: Throughputs with different thresholds

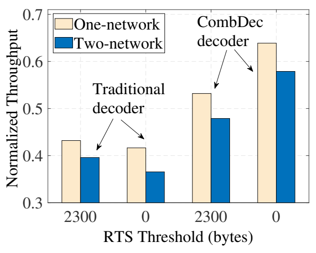


Figure 20: Throughputs in collocated networks.

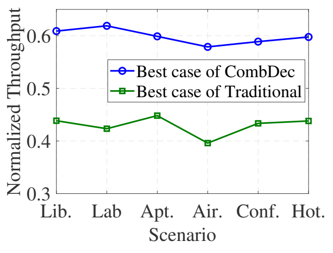


Figure 21: Throughputs in collocated networks.

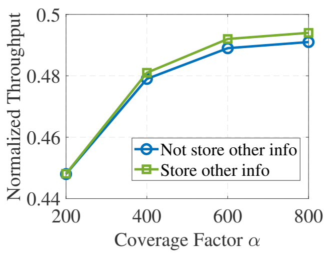


Figure 22: Storing information of other network.

of locations 1–8 to send data to the AP for the first round of experiments, and then let node 1 send data and node 2 send RTS for the second round. The first and second rounds represent the scenarios in which the receiving power of RTS is greater than and less than that of data, respectively. We consider the collision is resolved when both RTS and data packets are decoded successfully. Figure 16 depicts the success probability of collision resolution as node 2's location changes. The figure shows that generally, the success probability is higher when the receiving power of RTS is higher than that of data. This is because CombDec first treats any data packet as the noise to recover any RTS packet from the receiving signal. The results demonstrate that CombDec, primarily designed to handle RTS-only collisions, is capable of resolving RTS-data collisions in some scenarios.

5.3 Network Performance Evaluation

We then evaluate the benefits of CombDec for the network performance. Note that it is impossible to measure the network performance with CombDec under different setups at the same time because the resolution of a collision directly affects follow-on network dynamics. We have to measure the performance under different setups over non-overlapping measurement periods. Therefore, we conduct experiments during off-business hours to minimize the impact of environmental factors on different measurement periods.

5.3.1 Single Network Scenario

We first consider a single-network scenario: 12 nodes are placed at locations 0–11, in which the AP is at location 8, as shown in Figure 10. The network does not run in the MU-MIMO mode (i.e., the AP does not transmit data via MU-MIMO to multiple stations in the downlink). The RTS threshold is set to be 2,300 bytes for all nodes (i.e., RTS is triggered only when a data packet to be transmitted has a length over 2,300 bytes). The value of 2,300 is typical for today's WiFi products (e.g., the default value in Cisco APs is 2347 [9]).

Throughput Improvement: Figure 17 demonstrates the comparisons of normalized throughputs under traditional 802.11 decoding and CombDec. Note that the throughput performance is always measured at the AP. We can see that CombDec is able to uniformly boost the performance of traditional 802.11 decoding. For example, the normalized throughput for the airport scenario increases from 0.43 to 0.53, leading to a throughput gain ratio of (0.53 - 0.43)/0.43 = 23.3%. In all different scenarios, we observe that the throughput gain ratio under CombDec is 11.6%–23.3%.

Improvement by Tuning α and γ: We aim to find if we can improve the performance by tuning α and γ, which are important factors to balance the performance and complexity. Figure 18 shows the normalized throughputs for various α and γ values in the airport scenario. The figure shows that keeping increasing α and decreasing γ do not always lead to evident improvement. For example, when α goes from 600 to 800 and γ decreases from 20 to 10, the throughput under CombDec only increases from 0.491 to 0.494.

Improvement by Reducing RTS Threshold: It is still possible to further improve the network performance. Our observation is that traditionally, an RTS collision is considered not resolvable; therefore, many WiFi devices are conservative to set the RTS threshold. The transmission of a data packet triggers an RTS transmission only when its data size is greater than the threshold. In all previous experiments, the threshold is set as a typical value of 2,300 for today's networks. Now CombDec has the capability of decoding RTS collisions; therefore, it can be beneficial to encourage more RTS transmissions by reducing the threshold. Figure 19 compares the normalized throughputs under traditional 802.11 decoding and CombDec for different RTS thresholds. The figure shows that reducing the threshold generally decreases the throughput performance under traditional decoding; however and interestingly, it substantially boosts the performance under CombDec. The best case for traditional decoding is to set the threshold as 2000–2500, resulting in a normalized throughput of 0.456. By contrast, the best case for CombDec is to remove the threshold and let everyone always send RTS before data, yielding a higher throughput of 0.657. The throughput gain ratio is computed as (0.657-0.456)/0.456 = 44.08%. This encouraging result shows that CombDec has an immediate impact on today's practice of setting the RTS threshold, and significantly reducing this threshold can push WiFi towards a collision-free environment.

5.3.2 Collocated Networks

Next, we place a new network close to the single network used in previous experiments. In the new network, 8 nodes are placed at locations 12–19 and the AP is at location 15, as shown in Figure 10. The two networks use the same frequency and thus interfere with each other. We call the APs in the original and new networks AP 1 and AP 2, respectively.

Figure 20 shows the throughput performance under different settings in the airport scenario. In Figure 20, the one-network performance is the performance measured at AP 1 in the previous single-network scenario (without the new network); and the two-network performance is measured as the average of the throughputs measured at AP 1 and AP 2. We can observe from Figure 20 that when the new network is placed, the throughput performance degrades due to mutual interference. CombDec still performs better than traditional 802.11 decoding. In the two-network scenario, the best case for CombDec is setting the RTS threshold to 0 (which is also beneficial to solving the hidden terminal problem), yielding a throughput of 0.579; and the best case for traditional decoding has a throughput of 0.396. The throughput gain ratio is thus (0.579 - 0.395)/0.395 = 46.6%, which is also a substantial throughput improvement. Figure 21 compares the best case throughput performance between CombDec (removing the RTS threshold) and traditional coding (setting the threshold to 2,300) in different scenarios. It can be seen that the throughput gain ratio of CombDec is 33.6% – 46.2%.

As discussed in Section 3, CombDec is designed to only store information of its own network. In the two-network scenario, it is possible to enhance the performance of CombDec by letting (α,β)-construction store the information (including MAC addresses, NAVs, and RTS rates) of the other network. Figure 22 shows that storing other network information can further yet slightly improve the throughput performance of CombDec with a fairly large α.

6 Related Work

Interference Cancelation and Mitigation: In the literature, successive interference cancelation (SIC) has been proposed to decode collisions by using either pre-coded signatures or different receiving powers [15, 33, 35, 36, 39, 40, 53, 66]. The time offsets in different packet collisions (e.g., in the presence of hidden terminals) has also been leveraged to resolve collisions [32, 42, 63]. In addition, interference cancelation was widely studied in the full-duplex mode [20,22,23,56,66]. In cross-technology communication, corrupted packets may also be decoded by detecting the interference type [21,37]. A number of studies have also proposed interference alignment and nulling with or without channel state information [43,46]. These approaches cannot be readily adapted to regular WiFi scenarios considered in this paper, where RTS packets collide at the beginning of each transmission.

Multi-user Detection: CombDec is related to multi-user detection that attempts to decode multiple users' signals from the overlapped signal [25, 44, 61]. In cellular networks, CDMA has been widely used to assign distinct spread spectrum codes to different users [60]. However, there is no such code design in RTS packets. Constructive interference [25] is able to receive multiple synchronized transmissions. Nevertheless, it requires all packets have the same content, which is impossible for RTS signals. The work in [51] applied the time division technique to the byte level such that multiple users can share the same packet. This method needs a strict coordination among all users. A multi-user system is built in [44] through sharing multiple channels to users who are allowed to duplicate the signal into these channels. Applying these designs to WiFi requires modification of the standard; in contrast, CombDec is a non-invasive design.

Improving WiFi Performance: Substantial efforts have been devoted to improving the WiFi link performance [14, 16,30,34,48,49,55,57,65]. For example, [14,30,57] focused on optimizing the user selection algorithm in MU-MIMO and [16, 54, 65] aimed to improve the beamforming related techniques. Different algorithms were also investigated to improve the rate adaptation in WiFi [34, 49]. Recently, a rapid picocell switching has also been proposed for wireless transit networks [55]. CombDec is orthogonal to these studies that aim to improve WiFi performance in different aspects. We show that CombDec makes it possible to resolve RTS collisions and pushes WiFi towards a collision-free environment.

7 Conclusion

This paper provides a systematic study to resolve RTS collisions in WiFi networks. Our core contribution is a new decoding system CombDec that uses (α,β)-construction, γ-decimation and sparse recovery to resolve RTS collisions. CombDec does not require changing the 802.11 standard and redefines the role of the RTS functionality in WiFi. We show via system implementation and extensive evaluation that CombDec has a beneficial impact on WiFi networks and substantially improves the throughput performance by 33.6% – 46.2% in various scenarios.

Acknowledgments

We would like to thank Dr. Romit Roy Choudhury for shepherding our paper and the anonymous reviewers for the insightful comments. The work at University of South Florida in this paper was supported in part by NSF under grants CNS-1553304 and CNS-1717969.

References

[1] Arris router setup. http://www.cktv.ru/files/modem/ARRIS_Router_Setup_Web_GUI_UG.pdf, 2012.

[2] How expensive is an operation on a cpu. https://streamhpc.com/blog/2012-07-16/how-expensive-is-an-operation-on-a-cpu/, 2012.

[3] k largest(or smallest) elements in an array. https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/, 2012.

[4] Linksys user guide. https://content.etilize.com/User-Manual/1026969914.pdf, 2014.

[5] Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. *IEEE Std 802.11*, 2016.

[6] Dlink user manual. http://files.dlink.com.au/Products/DSL-2740M/Manuals/DSL-2740M_A1_Manual_v1.00(DI).pdf, 2017.

[7] Tplink user guide. https://static.tp-link.com/2017/201712/20171212/1910012191_TL-WR902AC%203.0_UG.pdf, 2017.

[8] GL.iNET B1300 IPQ4018 dual band wifi router. https://www.alibaba.com/product-detail/GL-iNET-B1300-ipq4018-dual-band_60779146003.html, 2018.

[9] Advanced radio settings. https://www.cisco.com/assets/sol/sb/isa500_emulator/help/guide/ae1269129.html, 2019.

[10] Introduction to fft and ofdm. http://shodhganga.inflibnet.ac.in/bitstream/10603/42180/10/10_chapter%202.pdf, 2019.

[11] Linksys official support. https://www.linksys.com/us/support-product?pid=01t80000003ouh0AAA, 2019.

[12] The viterbi algorithm. http://www.cim.mcgill.ca/~latorres/Viterbi/va_alg.htm, 2019.

[13] ZBT apg222. https://www.alibaba.com/product-detail/newest-atheros-chipset-IPQ4018-802-11ac_60541163350.html, 2019.

[14] Narendra Anand, Jeongkeun Lee, Sung-Ju Lee, and Edward W Knightly. Mode and user selection for multi-user mimo wlans without csi. In *Proc. of IEEE INFOCOM*, 2015.

[15] Jeffrey G Andrews. Interference cancellation for cellular systems: a contemporary overview. *IEEE Wireless Communications*, 12:19–29, 2005.

[16] Oscar Bejarano, Roger Pierre Fabris Hoefel, and Edward W Knightly. Resilient multi-user beamforming wlans: Mobility, interference, and imperfect csi. In *Proc. of IEEE INFOCOM*, 2016.

[17] Emmanuel Candes and Justin Romberg. l1-magic: Recovery of sparse signals via convex programming. 4:14, 2005.

[18] Emmanuel J Candes, Justin K Romberg, and Terence Tao. Stable signal recovery from incomplete and inaccurate measurements. *Communications on pure and applied mathematics*, 59(8):1207–1223, 2006.

[19] Emmanuel J Candes and Terence Tao. Decoding by linear programming. *IEEE transactions on Information Theory*, 51, 2005.

[20] Bo Chen, Yue Qiao, Ouyang Zhang, and Kannan Srinivasan. Airexpress: Enabling seamless in-band wireless multi-hop transmission. In *Proc. of ACM MobiCom*, 2015.

[21] Gonglong Chen, Wei Dong, Zhiwei Zhao, and Tao Gu. Towards accurate corruption estimation in zigbee under cross-technology interference. In *Proc. of IEEE ICDCS*, 2017.

[22] Lu Chen, Fei Wu, Jiaqi Xu, Kannan Srinivasan, and Ness Shroff. Bipass: Enabling end-to-end full duplex. In *Proc. of ACM MobiCom*, 2017.

[23] Jung Il Choi, Mayank Jain, Kannan Srinivasan, Phil Levis, and Sachin Katti. Achieving single channel, full duplex wireless communication. In *Proc. of ACM MobiCom*, 2010.

[24] Thomas M Cover and Joy A Thomas. *Elements Of Information Theory*. John Wiley & Sons, 2012.

[25] Manjunath Doddavenkatappa, Mun Choon Chan, Ben Leong, et al. Splash: Fast data dissemination with constructive interference in wireless sensor networks. In *Proc. of NSDI*, 2013.

[26] David L Donoho. Compressed sensing. *IEEE Transactions on information theory*, 52:1289–1306, 2006.

[27] David L Donoho, Michael Elad, and Vladimir N Temlyakov. Stable recovery of sparse overcomplete representations in the presence of noise. *IEEE Transactions on Information Theory*, 52:6–18, 2006.

[28] Simon Foucart and Holger Rauhut. *A mathematical introduction to compressive sensing*, volume 1. Birkhäuser Basel, 2013.

[29] Jim Geier. Wi-Fi: Define minimum SNR values for signal coverage. *Enterprise Networking Planet: Standards & Protocols*, 2008.

[30] Yasaman Ghasempour and Edward W Knightly. Decoupling beam steering and user selection for scaling multi-user 60 ghz wlans. In *Proc. of ACM MobiHoc*, 2017.

[31] Andrea Goldsmith. *Wireless communications*. Cambridge university press, 2005.

[32] Shyamnath Gollakota and Dina Katabi. *Zigzag decoding: combating hidden terminals in wireless networks*. 2008.

[33] Shyamnath Gollakota, Samuel David Perli, and Dina Katabi. Interference alignment and cancellation. In *Proc. of ACM SIGCOMM*, 2009.

[34] Aditya Gudipati and Sachin Katti. Strider: Automatic rate adaptation and collision handling. In *Proc. of ACM SIGCOMM*, 2011.

[35] Aditya Gudipati, Stephanie Pereira, and Sachin Katti. Automac: Rateless wireless concurrent medium access. In *Proc. of ACM MobiCom*, 2012.

[36] Daniel Halperin, Thomas Anderson, and David Wetherall. Taking the sting out of carrier sense: interference cancellation for wireless lans. In *Proc. of ACM MobiCom*, 2008.

[37] Anwar Hithnawi, Su Li, Hossein Shafagh, James Gross, and Simon Duquennoy. Crosszig: combating cross-technology interference in low-power wireless networks. In *Proc. of IEEE IPSN*, 2016.

[38] Piotr Indyk. Explicit constructions for compressed sensing of sparse signals. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–33. Society for Industrial and Applied Mathematics, 2008.

[39] Sachin Katti, Shyamnath Gollakota, and Dina Katabi. Embracing wireless interference: Analog network coding. *Proc. of ACM SIGCOMM*, 2007.

[40] Sachin Katti, Hariharan Rahul, Wenjun Hu, Dina Katabi, Muriel Médard, and Jon Crowcroft. Xors in the air: Practical wireless network coding. In *Proc. of ACM SIGCOMM*, 2006.

[41] Song Min Kim and Tian He. Freebee: Cross-technology communication via free side-channel. In *Proc. of ACM MobiCom*, 2015.

[42] Linghe Kong and Xue Liu. mzig: Enabling multi-packet reception in zigbee. In *Proc. of ACM MobiCom*, 2015.

[43] Swarun Kumar, Diego Cifuentes, Shyamnath Gollakota, and Dina Katabi. Bringing cross-layer mimo to today's wireless lans. In *Proc. of ACM SIGCOMM*, 2013.

[44] Tianji Li, Mi Kyung Han, Apurv Bhartia, Lili Qiu, Eric Rozner, Yin Zhang, and Brad Zarikoff. Crma: Collision-resistant multiple access. In *Proc. of ACM MobiCom*, 2011.

[45] Zhijun Li and Tian He. Webee: Physical-layer cross-technology communication via emulation. In *Proc. of ACM MobiCom*, 2017.

[46] Kyle Miller, Atresh Sanne, Kannan Srinivasan, and Sriram Vishwanath. Enabling real-time interference alignment: Promises and challenges. In *Proc. of ACM MobiHoc*, 2012.

[47] Renato DC Monteiro and Ilan Adler. Interior path following primal-dual algorithms. part i: Linear programming. *Mathematical programming*, 44(1-3):27–41, 1989.

[48] Peshal Nayak, Michele Garetto, and Edward W Knightly. Multi-user downlink with single-user uplink can starve tcp. In *Proc. of IEEE INFOCOM*, 2017.

[49] Jonathan Perry, Peter A Iannucci, Kermin E Fleming, Hari Balakrishnan, and Devavrat Shah. Spinal codes. In *Proc. of ACM SIGCOMM*, 2012.

[50] Qualcomm. Qualcomm ipq4018 soc. `https://www.qualcomm.com/products/ipq4018`, 2019.

[51] Sudipta Saha and Mun Choon Chan. Design and application of a many-to-one communication protocol. In *Proc. of IEEE INFOCOM*, 2017.

[52] I Richard Savage. Probability inequalities of the tchebycheff type. *Journal of Research of the National Bureau of Standards-B. Mathematics and Mathematical Physics B*, 65:211–222, 1961.

[53] Souvik Sen, Naveen Santhapuri, Romit Roy Choudhury, and Srihari Nelakuditi. Successive interference cancellation: A back-of-the-envelope perspective. In *Proc. of ACM SIGCOMM*, 2010.

[54] Clayton Shepard, Hang Yu, Narendra Anand, Erran Li, Thomas Marzetta, Richard Yang, and Lin Zhong. Argos: Practical many-antenna base stations. In *Proc. of ACM MobiCom*, 2012.

[55] Zhenyu Song, Longfei Shangguan, and Kyle Jamieson. Wi-fi goes to town: Rapid picocell switching for wireless transit networks. In *Proc. of ACM SIGCOMM*, 2017.

[56] Karthikeyan Sundaresan, Mohammad Khojastepour, Eugene Chai, and Sampath Rangarajan. Full-duplex without strings: Enabling full-duplex with half-duplex clients. In *Proc. of ACM MobiCom*, 2014.

[57] Sanjib Sur, Ioannis Pefkianakis, Xinyu Zhang, and Kyu-Han Kim. Practical mu-mimo user selection on 802.11 ac commodity networks. In *Proc. of ACM MobiCom*, 2016.

[58] George Turin. An introduction to matched filters. *IRE transactions on Information Theory*, 6:311–329, 1960.

[59] Sergio Verdu et al. *Multiuser detection*. Cambridge university press, 1998.

[60] Andrew J Viterbi and Andrew J Viterbi. *CDMA: principles of spread spectrum communication*, volume 122. Addison-Wesley Reading, MA, 1995.

[61] Yin Wang, Yunhao Liu, Yuan He, Xiang-Yang Li, and Dapeng Cheng. Disco: Improving packet delivery via deliberate synchronized constructive interference. *IEEE Transactions on Parallel & Distributed Systems*, pages 713–723, 2015.

[62] Allen Y Yang, Zihan Zhou, Arvind Ganesh Balasubramanian, S Shankar Sastry, and Yi Ma. Fast l1-minimization algorithms for robust face recognition. *IEEE Transactions on Image Processing*, 22:3234–3246, 2013.

[63] Junmei Yao, Tao Xiong, and Wei Lou. Beyond the limit: A fast tag identification protocol for rfid systems. *Pervasive and Mobile Computing*, 21:1–18, 2015.

[64] Xinyu Zhang and Kang G Shin. Cooperative carrier signaling: Harmonizing coexisting wpan and wlan devices. *IEEE/ACM Transactions on Networking (TON)*, 21:426–439, 2013.

[65] Anfu Zhou, Teng Wei, Xinyu Zhang, Min Liu, and Zhongcheng Li. Signpost: Scalable mu-mimo signaling with zero csi feedback. In *Proc. of ACM MobiHoc*, 2015.

[66] Wenjie Zhou, Tanmoy Das, Lu Chen, Kannan Srinivasan, and Prasun Sinha. Basic: backbone-assisted successive interference cancellation. In *Proc. of ACM MobiCom*, 2016.

APPENDIX A

We analyze popular WiFi drivers and AP firmware to understand these practical constraints: (i) Linux kernel drivers: ath9k (Qualcomm/Atheros 802.11n chipsets), brcmsmac (Broadcom 802.11n chipsets), and iwlwifi (Intel 802.11n/ac chipsets); (ii) 802.11ac AP firmware in Linksys EA8500, EA9500, TP-Link AC1200, C5400, and AD7200.

In particular, ath9k allows the maximum length of a data payload to be either 8,192 or 65,535 bytes (aPSDU-MaxLength for 802.11n is 65,535) based on its version number (as shown in Listing 1); brcmsmac uses the maximum duration of 5,000 μs (aPPDUMaxTime for 802.11n is 10,000 μs) (as shown in Listing 2); iwlwifi allows the maximum duration to be 4,000 μs (aPPDUMaxTime for 802.11ac is 5,484), as the excerpted code shown in Listing 3. In addition, Linksys EA8500, EA9500, and TP-Link AC1200, C5400 and AD7200 share the same code: the maximum length of a data payload is 65,535 bytes (aPSDUMaxLength for 802.11ac is 4,692,480)(as shown in Listing 4).

Listing 1: Source code in Qualcomm/Atheros ath9k (802.11n)

```
/* hw.h */
...
#define ATH_AMPDU_LIMIT_MAX    (64 * 1024 - 1)
...
/* hw.c */
    ...
    if (AR_SREV_9160_10_OR_LATER(ah) ||
        AR_SREV_9100(ah))
       pCap->rts_aggr_limit = ATH_AMPDU_LIMIT_MAX;
    else
       pCap->rts_aggr_limit = (8 * 1024);
    ...
```

Listing 2: Source code in Broadcom brcmsmac (802.11n)

```
/* ampdu.c */
...
/* max dur of tx ampdu (in msec) */
#define AMPDU_MAX_DUR          5
...
    ampdu->dur = AMPDU_MAX_DUR;
...
```

Listing 3: Source code of Intel iwlwifi (802.11ac)

```
/* mvm/constants.h */
...
#define IWL_MVM_RS_AGG_TIME_LIMIT      4000
...
/* mvm/rs.c */
...
    lq_cmd->agg_time_limit =
        cpu_to_le16(IWL_MVM_RS_AGG_TIME_LIMIT);
...
```

Listing 4: The same source code in Linksys EA8500/EA9500 and TP-Link AC1200/C5400/AD7200.

```
/* include/linux/ieee80211.h */
...
/*
Maximum length of AMPDU that the STA can receive.
Length = 2 ^ (13 + max_ampdu_length_exp)-1 (octets)
*/
enum ieee80211_max_ampdu_length_exp {
    ...
    IEEE80211_HT_MAX_AMPDU_64K = 3
};
...
```

APPENDIX B

Our comprehensive data collections capture WiFi packet traces under a diversity of traffic load conditions over long time periods. For each transmitter in the packet traces, we compute the NAV distribution in its RTS packets. We find that the NAV distribution is highly patterned or uneven in its value space. Figure 23 shows the NAV distributions of top five devices that send the largest numbers of RTS packets in different measurement environments. We observe from Figure 23 that each device's NAV values almost concentrate in the small value regions. For example, in the lab scenario, 92.1% NAV values in RTS packets from device 3 is 156.

We also notice that, interestingly, WiFi devices from the same manufacturer (identified by their MAC addresses) do exhibit similar NAV distribution in their RTS packets. Figure 24 illustrates the distributions of all NAV values in RTS packets transmitted by devices from 15 common manufacturers. It is seen from Figure 24 that the distributions exhibit different patterns by manufacturers, mainly due to their distinct firmware designs. Similar to Figure 23, Figure 24 shows the NAV distributions are highly uneven and patterned with a small number of NAV values much more likely to show up in RTS packets than the others.

In addition to NAV, we also measure the data rates of RTS packets. Figure 25 depicts the distribution of RTS dta rates in different environments. We can see that a large number of devices adopt 12 Mbps and 24 Mbps data rates to send RTS. Furthermore, in the conference and hotel scenarios, most devices even only use the 24 Mbps data rate.

Based on the packet trace analysis, if we select the NAV values that are most likely in RTS packet from each device to construct the comb matrix **M**, we should be able to decrease the size of **M** at the cost of a small performance penalty.

APPENDIX C

Performance of γ-decimation: For the tooth vector set $\mathcal{M} = \{\mathbf{m}_i\}_{i\in[1,M]}$, where $M = |\mathcal{M}|$ and $\mathbf{m}_i \in \mathbb{C}^{L\times 1}$, γ-decimation selects M/γ tooth vectors with largest correlation values to form a new comb matrix. Denote by $\mathcal{M}'$ the set consisting of these selected M/γ tooth vectors by γ-decimation. Without loss of generality, we assume the collided signal $\mathbf{y}$ contains the first S tooth vectors, i.e., $\mathbf{m}_1, \cdots, \mathbf{m}_S$. In this section, notations are summarized as follows: (i) $o(1)$ denotes a function that converges to 0 as $L \to \infty$; (ii) $\mathsf{E}(\cdot)$ and $\mathsf{Var}(\cdot)$ denote the expectation and variance operators, respectively; (iii) for a complex number m, m^* is the complex conjugate of m, and $|m|$ is the magnitude of m. Now we state the following theorem to show the performance of γ-decimation:

Theorem 1 *Define event A as the event that $\mathbf{m}_1$, $\mathbf{m}_2$, $\cdots$, $\mathbf{m}_s$ are all selected by γ-decimation in $\mathcal{M}'$. Then, it holds that $\mathsf{P}(A) = 1 - o(1)$ (i.e., event A happens with high probability).*

Proof: We first normalize the correlation between comb matrix **M** and the received vector **y**. From (2), we have

$$\mathbf{z} = \frac{1}{L}\mathbf{M}^H\mathbf{y} = \frac{1}{L}\begin{bmatrix} \mathbf{m}_1^H\mathbf{m}_1 & \mathbf{m}_1^H\mathbf{m}_2 & \cdots & \mathbf{m}_1^H\mathbf{m}_M \\ \mathbf{m}_2^H\mathbf{m}_1 & \mathbf{m}_2^H\mathbf{m}_2 & \cdots & \mathbf{m}_2^H\mathbf{m}_M \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{m}_M^H\mathbf{m}_1 & \mathbf{m}_{M-1}^H\mathbf{m}_2 & \cdots & \mathbf{m}_M^H\mathbf{m}_M \end{bmatrix}\begin{bmatrix} g1 \\ g2 \\ \vdots \\ g_M \end{bmatrix}. \tag{4}$$

Let $\mathbf{z} = [z_1, z_2, \cdots, z_M]^H$. It holds that $\forall z_i \in \mathbf{z}$,

$$z_i = \frac{1}{L}\sum_{s=1}^{M} g_s \mathbf{m}_i^H \mathbf{m}_s = \frac{1}{L}\sum_{s=1}^{M}\sum_{k=1}^{L} g_s m_{i,k}^* m_{s,k}. \tag{5}$$

Because each tooth vector $\mathbf{m}_i$ has the random property by coding, for any entry $m_{j,i} \in \mathbf{m}_i$, we have $\mathsf{E}(m_{j,i}) = 0$ and let $\mathsf{E}(|m_{j,i}|^2) = \sigma^2$.

Since tooth vectors $\mathbf{m}_1, \mathbf{m}_2, \cdots, \mathbf{m}_S$ are the ones to be resolved, we know the first S members in $\mathbf{g}$ are not zeros. Define Y as

$$Y = \sum_{m=S+1}^{M} \mathbf{1}_{\{|z_m|>h\}}, \tag{6}$$

denoting the number of false alarms (i.e., noise exceeding the threshold), where h is a threshold, and $\mathbf{1}_{\{|z_m|>h\}}$ is the indicator function defined as

$$\mathbf{1}_{\{|z_m|>h\}} = \begin{cases} 1, & \text{if } |z_m| > h \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

To evaluate the performance of γ-decimation, we define another event

$$B = \left(\bigcap_{s=1}^{S}\{|z_s| > h\}\right)\bigcap\{Y \le (\gamma M - S)\},$$

where the first part denotes that the correlation values z_1, z_2, $\cdots, z_S$ are all above the given threshold h and the second part indicates that there are at most $(\gamma M - S)$ other correlation values above h. If event B happens, $\mathbf{m}_1$, $\mathbf{m}_2$, $\cdots$, $\mathbf{m}_S$ will be selected by γ-decimation and thus event A must happen. This means $\mathsf{P}(A|B) = 1$ and $\mathsf{P}(A) \ge \mathsf{P}(B)$. Therefore, according to Fréchet inequalities, we obtain

$$\begin{aligned} \mathsf{P}(A) &\ge \mathsf{P}\left(\left(\bigcap_{s=1}^{S}\{|z_s| > h\}\right)\bigcap\{Y \le (\gamma M - S)\}\right) \\ &\ge \mathsf{P}\left(\bigcap_{s=1}^{S}\{|z_s| > h\}\right) - \mathsf{P}\left(Y > (\gamma M - S)\right). \end{aligned} \tag{8}$$

From (8), we need two steps to finish the proof:

- **Step 1:** prove $\mathsf{P}\left(\bigcap_{s=1}^{S}\{|z_s| > h\}\right) = 1 - o(1)$.
- **Step 2:** prove $\mathsf{P}\left(Y > (\gamma M - S)\right) = o(1)$.

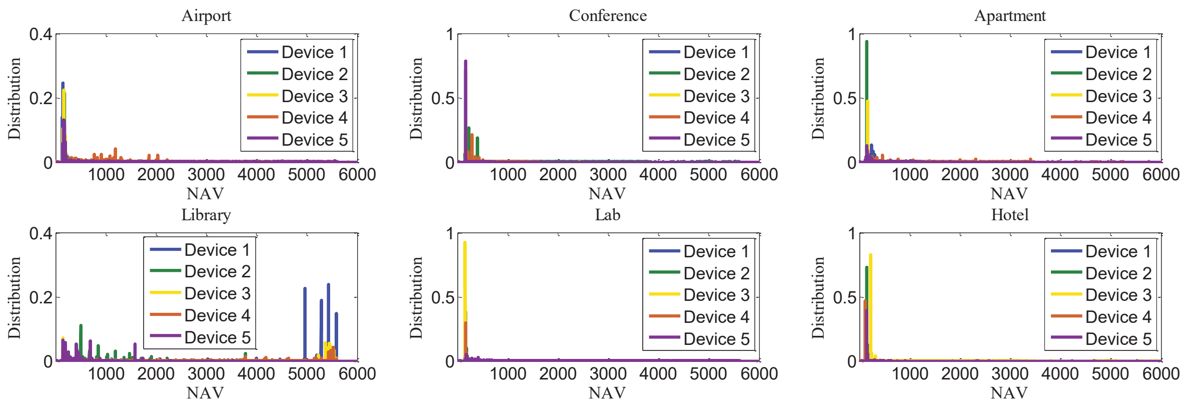


Figure 23: NAV distributions at different locations.

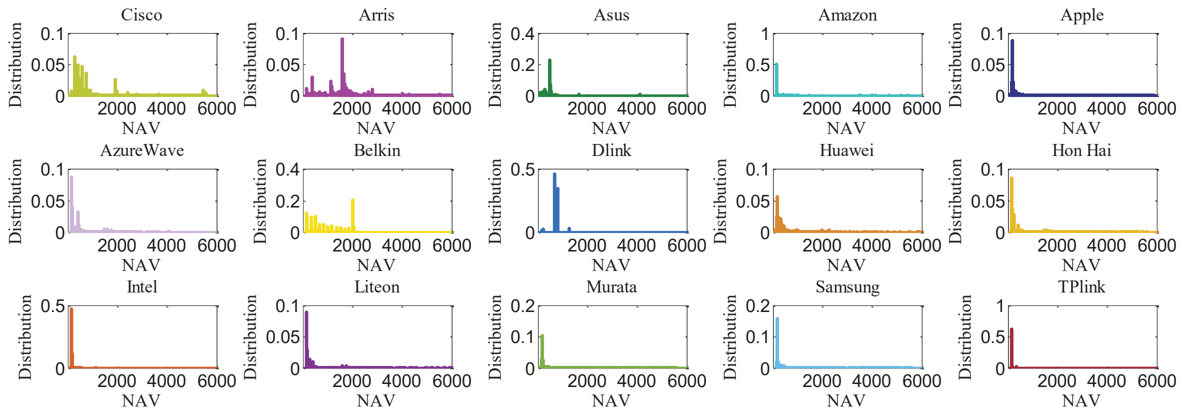


Figure 24: NAV distributions of different vendors.

Step 1:

By Fréchet inequalities, we have that

$$\mathsf{P}\left(\bigcap_{s=1}^{S}\{|z_s|>h\}\right)\geq\sum_{s=1}^{S}\mathsf{P}(|z_s|>h)-(S-1). \quad (9)$$

According to Cantelli's inequality [52], for $1\leq s\leq S$, we have the probability

$$\mathsf{P}(|z_s|>h)\geq 1-\frac{\mathsf{Var}(|z_s|)}{\mathsf{Var}(|z_s|)+(h-\mathsf{E}(|z_s|))^2}. \quad (10)$$

Next we derive $\mathsf{E}(|z_s|)$ and $\mathsf{Var}(|z_s^2|)$ respectively. Because $1\leq s\leq S$, without loss of generality, we consider the first element z_1. From (5), by leveraging Lemma 1, we have

$$\mathsf{E}(|z_1|)=\mathsf{E}(\frac{1}{L}g_1\mathbf{m}_1^H\mathbf{m}_1+\frac{1}{L}\sum_{s=2}^{S}g_s\mathbf{m}_1^H\mathbf{m}_s)=g_1\sigma^2, \quad (11)$$

and

$$\begin{aligned}\mathsf{E}(|z_1|^2)&=\mathsf{E}\left(\frac{1}{L^2}\left(\mathbf{m}_1^H\sum_{s=1}^{S}g_s\mathbf{m}_s\right)^2\right)\\&=\frac{1}{L^2}\mathsf{E}\left(\sum_{s=1}^{S}\sum_{k=1}^{L}g_sm_{1,k}^*m_{s,k}\sum_{s'=1}^{S}\sum_{k'=1}^{L}g_{s'}m_{1,k'}^*m_{s',k'}\right)\\&=\frac{1}{L^2}\sum_{s=1}^{S}\sum_{k=1}^{L}\sum_{s'=1}^{S}\left(\sum_{k'=1,k'\neq k}^{L}g_sg_{s'}\mathsf{E}(m_{1,k}^*m_{s,k})\right.\\&\qquad\left.\times\mathsf{E}(m_{1,k'}^*m_{s',k'})+g_sg_{s'}\mathsf{E}(m_{1,k}^*m_{s,k}m_{1,k}^*m_{s',k})\right).\\&=g_1^2\sigma^4+\frac{1}{L}\Xi_1,\end{aligned} \quad (12)$$

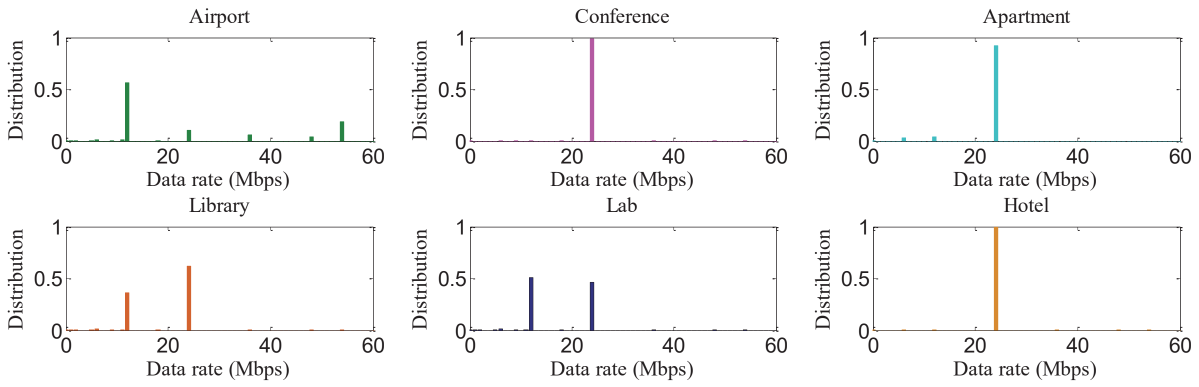


Figure 25: RTS data rate distributions.

where

$$\Xi_1 = \left(g_1^2(\mathsf{E}(|m_{1,k}|^4) - \sigma^4) + \mathsf{E}((m_{1,k}^*)^2)\mathsf{E}(m_{1,k}^2)\sum_{s=2}^{S} g_s^2 \right).$$

From (11) and (12), we obtain the variance of z_1 as

$$\mathsf{Var}(|z_1|) = \mathsf{E}(|z_1|^2) - (\mathsf{E}(|z_1|))^2 = \frac{1}{L}\Xi_1. \tag{13}$$

Replacing (13) into (10), we have

$$\mathsf{P}(|z_1| > h) \geq 1 - \frac{\Xi_1}{\Xi_1 + L(h - g_s\sigma^2)^2}. \tag{14}$$

Then, (9) can be rewritten as

$$\begin{aligned}\mathsf{P}\left(\bigcap_{s=1}^{S}\{|z_s| > h\}\right) &\geq 1 - \sum_{s=1}^{S}\frac{\Xi_s}{\Xi_s + L(h - g_s\sigma^2)^2} \\ &\geq 1 - S\frac{\Xi_{\max}}{\Xi_{\max} + L(h - g_{\max}\sigma^2)^2},\end{aligned} \tag{15}$$

where $\Xi_{\max} = \max\{\Xi_s\}_{s\in[1,S]}$, and $g_{\max} = \max\{g_s\}_{s\in[1,S]}$. When $L \to \infty$, the probability converges to 1.

Step 2:

Letting $y = \gamma M - S$, by Markov's inequality, we have

$$\mathsf{P}(Y > y) \leq \frac{1}{y}\mathsf{E}(Y), \tag{16}$$

then from (6), we have

$$\mathsf{E}(Y) = \mathsf{E}\left(\sum_{m=S+1}^{M} \mathbf{1}_{\{|z_m|>h\}}\right) = \sum_{m=S+1}^{M} \mathsf{P}(|z_m| > h). \tag{17}$$

According to Chebyshev's inequality, we can obtain

$$\mathsf{P}(||z_m| - \mathsf{E}(|z_m|)| > h) \leq \frac{\mathsf{Var}(|z_m|)}{h^2}. \tag{18}$$

Next, we derive $\mathsf{E}(|z_m|)$ and $\mathsf{Var}(|z_m|)$. Without loss of generality, we consider the last element z_M. Similarly, we have

$$\mathsf{E}(|z_M|) = \frac{1}{L}\mathsf{E}\left(\sum_{s=1}^{S}\sum_{k=1}^{L} g_s m_{M,k}^* m_{s,k}\right) = 0, \tag{19}$$

and

$$\begin{aligned}\mathsf{E}(|z_M|^2) &= \frac{1}{L^2}\mathsf{E}\left(\sum_{s=1}^{S}\sum_{k=1}^{L} g_s m_{M,k}^* m_{s,k}\right)^2 \\ &= \frac{1}{L^2}\sum_{s=1}^{S}\sum_{k=1}^{L}\sum_{s'=1}^{S} g_s g_{s'}\mathsf{E}(m_{M,k}^* m_{s,k} m_{M,k}^* m_{s',k}) \\ &= \frac{1}{L}\sum_{s=1}^{S} g_s^2\mathsf{E}((m_{M,k}^*)^2)\mathsf{E}(m_{s,k}^2).\end{aligned} \tag{20}$$

Let $\Psi_M = \sum_{s=1}^{S} g_s^2\mathsf{E}((m_{M,k}^*)^2)\mathsf{E}(m_{s,k}^2)$, then we can obtain

$$\mathsf{Var}(|z_M|) = \mathsf{E}(|z_M|^2) - (\mathsf{E}(|z_M|))^2 = \frac{1}{L}\Psi_M. \tag{21}$$

Replacing (19) and (21) into (18), we have

$$\mathsf{P}(|z_M| > h) \leq \frac{\Psi_M}{Lh^2}. \tag{22}$$

Thus (17) can be rewritten as

$$\mathsf{E}(Y) \leq \sum_{m=S+1}^{M}\frac{\Psi_M}{Lh^2} \leq (M - S - 1)\frac{\Psi_{\max}}{Lh^2}, \tag{23}$$

where $\Psi_{\max} = \max\{\Psi_m\}_{m\in[S+1,M]}$. Finally,

$$\mathsf{P}(Y > y) \leq \frac{1}{y}\mathsf{E}(Y) \leq \frac{(M - S - 1)\Psi_{\max}}{Lyh^2}. \tag{24}$$

When $L \to \infty$, $\mathsf{P}(Y > y)$ converges to 0, which completes the proof. □

Lemma 1 *Given tooth vectors $\mathbf{m}_i$ and $\mathbf{m}_j$, for all $m_{k,i} \in \mathbf{m}_i$ and $m_{s,j} \in \mathbf{m}_j$ satisfying $\mathsf{E}(m_{k,i}) = \mathsf{E}(m_{s,j}) = 0$ and $\mathsf{E}(|m_{k,i}|^2) = \mathsf{E}(|m_{s,j}|^2) = \sigma^2$, the following two statements are true: (i) if $i \neq j$, $\mathsf{E}(\frac{1}{L}\mathbf{m}_i^H\mathbf{m}_j) = 0$ and $\mathsf{E}(|\frac{1}{L}\mathbf{m}_i^H\mathbf{m}_j|^2) = \frac{1}{L}\sigma^4$; (ii) if $i = j$, $\mathsf{E}(\frac{1}{L}\mathbf{m}_i^H\mathbf{m}_i) = \mathsf{E}(\frac{1}{L}\mathbf{m}_j^H\mathbf{m}_j) = \sigma^2$ and $\mathsf{E}(|\frac{1}{L}\mathbf{m}_i^H\mathbf{m}_i|^2) = \frac{1}{L}(L-1)\sigma^4 + \frac{1}{L}3\sigma^4$.*

Proof: Let $z = \frac{1}{L}\mathbf{m}_i^H\mathbf{m}_j = \frac{1}{L}\sum_{k=1}^{L} m_{i,k}^* m_{j,k}$.

For statement (i), since $\mathsf{E}(m_{i,k}) = \mathsf{E}(m_{j,k}) = 0$, we have

$$\mathsf{E}(\frac{1}{L}\mathbf{m}_i^H\mathbf{m}_j) = 0.$$

Furthermore, as we know $\mathsf{E}(|m_{i,k}|^2) = \mathsf{E}(|m_{s,k}|^2) = \sigma^2$, we can obtain

$$\begin{aligned}\mathsf{E}(|\frac{1}{L}\mathbf{m}_i^H\mathbf{m}_j|^2) &= \frac{1}{L^2}\mathsf{E}\left(\sum_{k=1}^{L} m_{i,k}^* m_{j,k} \sum_{q=1}^{L} m_{i,q}^* m_{j,q}\right) \\ &= \frac{1}{L}\sigma^4\end{aligned} \tag{25}$$

For statement (ii), it is easy to know that

$$\mathsf{E}(\frac{1}{L}\mathbf{m}_i^H\mathbf{m}_j) = \mathsf{E}(\frac{1}{L}\mathbf{m}_i^H\mathbf{m}_i) = \sigma^2$$

and

$$\begin{aligned}\mathsf{E}(|\frac{1}{L}\mathbf{m}_i^H\mathbf{m}_i|^2) &= \mathsf{E}\left(\frac{1}{L}\sum_{k=1}^{L} m_{i,k}^* m_{i,k}\right)^2 \\ &= \frac{1}{L^2}L(L-1)\sigma^4 + \frac{1}{L^2}\sum_{k=1}^{L}\mathsf{E}(|m_{i,k}|^4) \\ &= \frac{1}{L}(L-1)\sigma^4 + \frac{1}{L}3\sigma^4.\end{aligned} \tag{26}$$

Therefore, we complete the proof. □

Plankton: Scalable network configuration verification through model checking

Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P. Brighten Godfrey, Matthew Caesar
University of Illinois at Urbana-Champaign

Abstract

Network configuration verification enables operators to ensure that the network will behave as intended, prior to deployment of their configurations. Although techniques ranging from graph algorithms to SMT solvers have been proposed, scalable configuration verification with sufficient protocol support continues to be a challenge. In this paper, we show that by combining equivalence partitioning with explicit-state model checking, network configuration verification can be scaled significantly better than the state of the art, while still supporting a rich set of protocol features. We propose Plankton, which uses symbolic partitioning to manage large header spaces and efficient model checking to exhaustively explore protocol behavior. Thanks to a highly effective suite of optimizations including state hashing, partial order reduction, and policy-based pruning, Plankton successfully verifies policies in industrial-scale networks quickly and compactly, at times reaching a 10000× speedup compared to the state of the art.

1 Introduction

Ensuring correctness of networks is a difficult and critical task. A growing number of network verification tools are targeted towards automating this process as much as possible, thereby reducing the burden on the network operator. Verification platforms have improved steadily in the recent years, both in terms of scope and scale. Starting from offline data plane verification tools like Anteater [19] and HSA [13], the state of the art has evolved to support real-time data plane verification [15, 12], and more recently, analysis of configurations [6, 5, 7, 1, 25].

Configuration analysis tools such as Batfish [6], ERA [5], ARC [7] and Minesweeper [1] are designed to take as input a given network configuration, a correctness specification, and possibly an "environment" specification, such as the maximum expected number of failures, external route advertisements, etc. Their task is to determine whether, under the given environment specification, the network configuration will always meet the correctness specification. As with most formal verification domains, the biggest challenge in configuration analysis is scalability. Being able to analyze the behavior of multiple protocols executing together is a nontrivial task. Past verifiers have used a variety of techniques to try to surmount this scalability challenge. While many of them sacrifice their correctness or expressiveness in the process, Minesweeper maintains both by modeling the network using SMT constraints and performing the verification using an SMT solver. However, we observe that this approach scales poorly with increasing problem size (4+ hours to check a 245-device network for loops, in our experiments). So, this paper addresses the following question: *Can a configuration verification tool have broad protocol support, and also scale well?*

We begin our work by observing that scalability challenges in configuration verification stem from two factors — the large space of possible packet headers, and the possibly diverse outcomes of control plane execution, particularly in the presence of failures. We believe that general purpose SAT/SMT techniques are not as well equipped to tackle these challenges as domain-specific techniques specifically designed to address them. In fact, these challenges have been studied extensively, in the domains of data plane verification and software verification. Data plane verification tools analyze the large header space to determine all possible data plane behaviors and check their correctness. Software verification techniques explore the execution paths of software, including distributed software, and identify undesirable executions that often elude testing. Motivated by the success of the analysis algorithms in these domains, we attempted to combine the two into a scalable configuration verification platform. The result — Plankton — is a configuration verifier that uses equivalence partitioning to manage the large header space, and explicit-state model checking to explore protocol execution. Thanks to these efficient analysis techniques, and an extensive suite of domain-specific optimizations, Plankton delivers consistently high verification performance.

Feature	Batfish	BagPipe	ARC	ERA	Minesweeper	Plankton
All data planes, including failures	○	○	◐	○	●	●
Support beyond specific protocols	●	○	○	●	●	●
Soundness when assumptions hold	●	●	●	◐*	●	●

* *For segmentation policies only*

Figure 1: Comparison of configuration verification systems

Our key contributions are as follows:

- We define a configuration verification paradigm that combines packet equivalence computation and explicit-state model checking.
- We develop optimizations that make the design feasible for networks of practical scale, including optimizations to reduce the space of event exploration, and techniques to improve efficiency of exploration.
- We implement a Plankton prototype with support for OSPF, BGP and static routing, and show experimentally that it can verify policies at practical scale (less than a second for networks with over 300 devices). Plankton outperforms the state of the art in all our tests, in some cases by as much as 4 orders of magnitude.

2 Motivation and Design Principles

Configuration verifiers take as input the configuration files for network devices, and combine them with an abstraction of the lower layers — the distributed control plane, which executes to produce data plane state, and the forwarding hardware that will use that state to process traffic. They may additionally take an *environment specification*, which describes interactions of entities external to the network, such as possible link failures, route advertisements, etc. Their task is to verify that under executions enabled by the supplied configuration and environment, correctness requirements are never violated. Configuration verifiers thus help ensure correctness of proposed configurations prior to deployment.

Figure 1 illustrates the current state of the art in configuration verification. As the figure shows, only Minesweeper [1] can reason about multiple possible converged data plane states of the network (e.g., due to topology changes or control plane non-determinism), while also having the ability to support more than just a specific protocol, and maintaining soundness of analysis. All other tools compromise on one or more key features. ARC [7], for example, uses graph algorithms to compute the multiple converged states enabled by failures, but only for shortest-path routing. As a result it cannot handle common network configurations such as BGP configurations that use LocalPref, any form of recursive routing, etc. The reason for the mismatch in Minesweeper's functionality in contrast to others is that it makes a different compromise. By using an SMT-based formulation, Minesweeper is able to achieve good data plane coverage and feature coverage, but pays the price in performance. As experiments show [2], Minesweeper scales poorly with network size, and is unable to handle networks larger than a few hundred devices in reasonable time. Our motivation for Plankton is simple — can we design a configuration verification tool without compromising scale or functionality?

Achieving our goal requires tackling two challenges: packet diversity and data plane diversity. Packet diversity, which refers to the large space of packets that needs to be checked, is easier to handle. We leverage the notion of *Packet Equivalence Classes* (PECs), which are sets of packets that behave identically. Using a trie-based technique similar to VeriFlow [15], we compute PECs as a partitioning of the packet header space such that the behavior of all packets in a PEC remains the same throughout Plankton's exploration. A more interesting aspect of PECs is how to handle dependencies across PECs without compromising performance. In Plankton, this is done by a dependency-aware scheduler designed to maximize independent analysis (§ 3.2).

Data plane diversity refers to the complexity of checking every possible converged data plane that an input network configuration may produce. It is the task of the control plane model to ensure coverage of these possible outcomes. Simulation-based tools (the best example being Batfish [6]) execute the system only along a single non-deterministic path, and can hence miss violations in networks that have multiple stable convergences, such as certain BGP configurations. ARC's graph-based approach accounts for possible failures, but can support only shortest-path routing. In order to overcome these shortcomings, Minesweeper, the current state of the art in terms of functionality, uses an SMT solver to search through possible failures and non-deterministic protocol convergence, to find any converged state that represents a violation of network correctness.

A key intuition behind our approach is that the *generic search technique employed by SMT solvers makes the core of the configuration verification problem much more difficult than it has to be*. Network control planes operate using simple algorithms which can not only be easily modeled in software, but can also find a protocol's outcome much more quickly than general-purpose SMT solving. In fact, the common case is that the control plane computes some variant of shortest or least-cost paths. To illustrate this point, we implemented simple single-source shortest path solvers in SMT (Z3) and a model checker (SPIN). The SMT formulation is implemented as constraints on the solution, while the model checker explores execution paths of the Bellman-Ford algorithm; and in this simplistic case the software has deterministic execution. The result is that the model checker approach is similar to a normal execution of software, and is around 12,000× faster even in a moderate-sized fat tree network of $N = 180$ nodes (Figure 2).

Of course, this is intentionally a simplistic, fully-

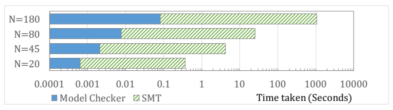


Figure 2: Comparison of two ways to compute shortest paths.

deterministic case with simple implementations. The point is that the model checking approach begins with a huge advantage — so huge that it could explore many non-deterministic execution paths and still outperform SMT. This leads to our next key intuition: *the effect of non-determinism is important, but the amount of "relevant" non-determinism is limited*. Networks can experience "relevant" non-determinism like the choice of what failures occur, and protocol execution; as well as "irrelevant" non-determinism like message timing that ultimately doesn't affect the outcome. Configurations and protocols are usually designed to keep the outcome mostly deterministic, with non-deterministic branch points ultimately leading to one or a small number of different converged states.

Motivated by this intuition, we create a control plane model that incorporates the possible non-deterministic behaviors, but we also implement optimizations so that when the *outcome* of the executions is actually deterministic, the "irrelevant" non-determinism is pruned enough that performance is comparable to simulation. This model is exhaustively explored by SPIN, a model checker designed for Promela programs. SPIN performs a depth-first search on the state space of the program, looking for states that violate the policy being checked. We further assist SPIN through optimizations that minimize the size of individual states, thus making the traversal process more efficient. Thanks to these two types of optimizations, Plankton achieves our goal of scalable, general-purpose configuration verification.

3 Plankton Design

We now present Plankton's design, illustrated in Figure 3.

3.1 Packet Equivalence Classes

The first phase in Plankton's analysis is the computation of Packet Equivalence Classes. As we discussed in § 2, Plankton uses a trie-based technique inspired by VeriFlow. The trie in Plankton holds prefixes obtained from the configuration, including any prefixes that are advertised (explicitly or automatically), any prefixes appearing in route maps, any static routes, etc. Each prefix in the trie is associated with a config object, that describes any configuration information that is specific to that prefix. For example, consider Figure 4, which illustrates a highly simplified example where the prefixes `128.0.0.0/1` and `192.0.0.0/2` are being advertised over OSPF in a topology with 3 devices. The trie holds three config objects — the default, and one for each advertised prefix.

Once the trie is populated, Plankton performs a recursive traversal, simultaneously keeping track of where the prefix boundaries define division of the header space. For each known partition, it stores the most up-to-date network-wide config known. When the end of any new prefix is reached, the config object that is associated with it is merged with the network-wide config for the partition that

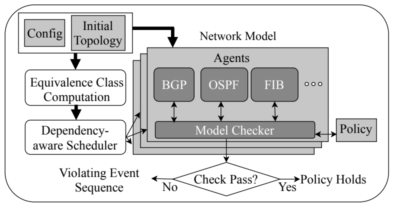


Figure 3: Plankton design

denotes the prefix. In our simple example, the traversal produces three classes defined by ranges — `[192.0.0.0, 255.255.255.255]` with two nodes originating prefixes, `[128.0.0.0, 191.255.255.255]` with only one origin, and `[0.0.0.0, 127.255.255.255]` without any node originating any prefix. As the example shows, each PEC-specific configuration computed this way will still include information about the original prefixes contributing to the PEC. Storing these prefixes may seem redundant. However, note that even within a PEC, the lengths of the prefixes that get advertised or get matched in route filters play a role in decision making.

3.2 Dependency-aware Scheduling

It is tempting to believe that Packet Equivalence Classes could be analyzed fully independently of each other, and that an embarrassingly parallel scheme could be used in the verification process. While this is indeed true sometimes, there can often be dependencies between various PECs. For example, consider a network that is running iBGP. For the various peering nodes to be able to communicate with each other, an underlying routing protocol such as OSPF should first establish a data plane state that forwards packets destined to the devices involved in BGP. In such a network, the manner in which OSPF determines the forwarding behavior for the device addresses will influence the routing decisions made in BGP. In other words, the PECs that are handled by BGP depend on the PECs handled by OSPF. In the past, configuration verification tools have either ignored such cases altogether, or, in the case of Minesweeper, modeled these classes simultaneously. Specifically, for a network of with n routers running iBGP, Minesweeper creates $n+1$ copies of the network, searching for the converged solution for the n loopback addresses and also BGP. Effectively, this turns the verification problem into one quadratically larger than the original. Given that configuration verification scales significantly worse than linearly in input size, such a quadratic increase in input size often makes the problem intractable.

Plankton goes for a more surgical approach. Once the PECs are calculated, Plankton identifies dependencies between the Packet Equivalence Classes, based on recursive routing entries, BGP sessions, etc. The dependencies are stored as a directed graph, whose nodes are the PECs, and directed edges indicate which PECs depend on which others. In order to maximize parallelism in verification runs across PECs, a dependency-aware scheduler first identifies strongly

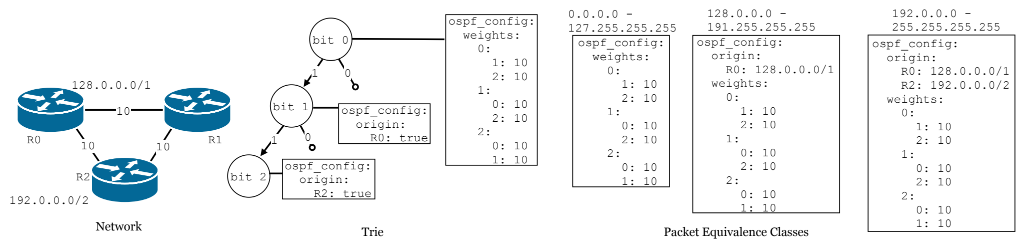

128.0.0.0/1
10
R0
R1
10
10
R2
192.0.0.0/2
Network
bit 0
1
0
bit 1
1
0
bit 2
ospf_config: weights: 0: 1: 10 2: 10 1: 0: 10 2: 10 2: 0: 10 1: 10
ospf_config: origin: R0: true
ospf_config: origin: R2: true
Trie
0.0.0.0 - 127.255.255.255
128.0.0.0 - 191.255.255.255
192.0.0.0 - 255.255.255.255
ospf_config: origin: R0: 128.0.0.0/1
ospf_config: origin: R0: 128.0.0.0/1 R2: 192.0.0.0/2
Packet Equivalence Classes

Figure 4: Packet Equivalence Class computation

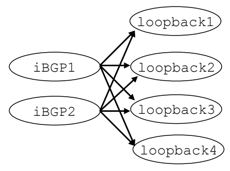


Figure 5: PEC Dependency Graph

connected components in the graph. These SCCs represent groups of PECs that are mutually dependent, and hence need to be analyzed simultaneously through a single verification run. In addition, if an SCC is reachable from another, it indicates that the upstream SCC can be scheduled for analysis only after the one downstream has finished. Each verification run is a separate process. For an SCC S, if there is another SCC S' that depends on it, Plankton forces all possible outcomes of S to be written to an in-memory filesystem (S will always gets scheduled first). Outcomes refer to every possible converged state for S, together with the non-deterministic choices made in the process of arriving at them. When the verification of S' gets scheduled, it reads these converged states, and uses them when necessary.

Minesweeper's technique of replicating the network roughly corresponds to the case where all PECs fall into a single strongly connected component. We expect this to almost never be the case. In fact, in typical cases, the SCCs are likely to be of size 1, meaning that every PEC can be analyzed in isolation, with ordering of the runs being the only constraint*. For example, Figure 5 illustrates the dependency graph for a typical case where two PECs are being handled in iBGP on a network with 4 different routers. The only constraint in this case is that the loopback PECs should be analyzed before the iBGP PECs can start. In such cases, Plankton's keeps the problem size significantly smaller, and maximizes the degree of parallelism that can be achieved.

When a PEC needs the relevant converged states of past PECs for its exploration, the non-deterministic choices may need to be coordinated across all these PECs. In particular, consider the choice of link failures: if we hypothetically executed one PEC assuming link L has failed and another PEC assuming L is working, the result represents an invalid execution of the overall network. Therefore, our current prototype matches topology changes across explorations. A second class of non-deterministic choices is protocol non-determinism. In our experiments, we have not seen cases of protocol non-determinism that requires matching across PECs. OSPF by its nature has deterministic outcomes, but on networks which have non-determinism in their internal routing (e.g., non-deterministically configured BGP for internal routing) and where message timing is correlated across PECs (e.g., via route aggregation), the system would need to coordinate this non-determinism to avoid false positives.

*An example where the SCCs are bigger than one PEC is the contrived case where there exists a static route for destination IP A whose next hop is IP B, but another static route for destination IP B whose next hop is IP A.

3.3 Explicit-state Model Checker

The explicit state model checker SPIN [9] provides Plankton its exhaustive exploration ability. SPIN verifies models written in the Promela modeling language, which has constructs to describe possible non-deterministic behavior. SPIN's exploration is essentially a depth-first search over the possible states of the supplied model. Points in the execution where non-deterministic choices are made represent branching in the state space graph.

Plankton's network model is essentially an implementation of the control plane in Promela. Our current implementation supports OSPF, BGP and static routing. Recall from § 3.1 that Plankton partitions the header space into Packet Equivalence Classes. For each SCC, Plankton uses SPIN to exhaustively explore control plane behavior. In order to improve scalability, Plankton also performs another round of partitioning by executing the control plane for each prefix in the PEC separately. This separation of prefixes is helpful in simplifying the protocol model. However, it does limit Plankton's support for route aggregation. While features such as a change in the routing metric can be supported, if there is a route map that performs an exact match on the aggregated prefix, it will not apply to the more specific routes, which Plankton models. Once the converged states of all relevant prefixes are computed, a model of the FIB combines the results from the various prefixes and protocols into a single network-wide data plane for the PEC.

In what follows, we present Plankton's network model that will be executed by SPIN. We will initially present a simple, unoptimized model, which is functionally correct but has significant non-determinism that is irrelevant to finding different converged states. Subsequently, in § 4, we discuss how Plankton attempts to minimize irrelevant non-determinism, making the execution of the deterministic fragments of the control plane comparable to simulation.

3.4 Abstract Protocol Model

To define a control plane suitable for modeling real world protocols such as BGP and OSPF, we look to the technique used by Minesweeper wherein the protocols were modeled as instances of the stable paths problem. Along similar lines, we consider the Simple Path Vector Protocol [8], which was originally proposed to solve the stable paths problem. We first extend SPVP to support some additional features that we wish to model. Based on this, we construct a protocol we call the *Reduced Path Vector Protocol*, which we show to be sufficient to correctly perform model checking, if we are interested only in the converged states of the network. We use RPVP as the common control plane protocol for Plankton. We begin with a brief overview of SPVP, highlighting our extensions to the protocol. Appendix A contains the full details of the protocol and our extensions.

3.4.1 SPVP and its extension

SPVP is an abstract model of real-world BGP, replacing the details of BGP configurations with abstract notions of import/export filters and ranking functions. For each node n and peer n', the import and export filters dictate which advertisements (i.e. the advertiser's best path to the *origin*) n can receive from and send to n', respectively. The ranking function for n dictates the preference among all received advertisements. These notions can be inferred from real-world configurations.

We slightly extend the original SPVP [8] to support more features of BGP. The extensions are as follows: we allow for multiple origins instead of a single one; the ranking function can be a *partial order* instead of a total one to allow for time based tie breaking; and to be able to model iBGP, we allow the ranking function of any node to change at any time during the execution of protocol.

It is well known that there are configurations which can make SPVP diverge in some or all execution paths. However, our goal is only to check the forwarding behavior in the converged states, through explicit-state model checking. So, we define a much simpler model that can be used, without compromising the soundness or completeness of the analysis (compared to SPVP).

3.4.2 Reduced Path Vector Protocol (RPVP)

We now describe RPVP, which is specifically designed for explicit-state model checking of the converged states of the extended SPVP protocol.

In RPVP, the message passing model of SPVP is replaced with a shared memory model. The network state only consists of the values of the best known path of each node at each moment (best-path). In the initial state, the best path of all nodes is $\perp$, except origins, whose best path is ε. At each step, the set of all enabled nodes (E) is determined (Algorithm 1, line 5). A node n is considered enabled if either i) the current best path p of n is invalid, meaning that the next hop in p has a best path that is not a prefix of p.

$$\text{invalid}(n) \triangleq \text{best-path}(\text{best-path}(n).\text{head}) \neq \text{best-path}(n).\text{rest}$$

Algorithm 1 RPVP

```
1:  procedure RPVP(:)
2:      Init : $\forall n \in N - Origins.\ \text{best-path}(n) \leftarrow \perp$
3:      Init : $\forall o \in Origins.\ \text{best-path}(o) = \varepsilon$
4:      while true do:
5:          $E \leftarrow \{n \in N \mid \text{invalid}(n) \vee \exists n' \in \text{peers}(n).\ \text{can-update}_{\text{n}}(n')\}$
6:          if $E = \emptyset$ then:
7:              break
8:          end if
9:          $n \leftarrow \text{nondet-pick}(E)$
10:         if $\text{invalid}(n)$ then
11:             $\text{best-path}(n) \leftarrow \perp$
12:         end if
13:         $U \leftarrow \text{best}(\{n' \in \text{peers}(n) \mid \text{can-update}_{\text{n}}(n')\})$
14:         $n' \leftarrow \text{nondet-pick}(U)$
15:         $p \leftarrow \text{import}_{n,n'}(\text{export}_{n',n}(\text{best-path}(n')))$
16:         $\text{best-path}(n) \leftarrow p$
17:     end while
18: end procedure
```

Or ii) there is a node n' among the peers of n that can produce an advertisement which will change the current best path of n. In other words, n' has a path better than the current best path of n, and the path is acceptable according to the export and import policies of n' and n respectively.

$$\text{can-update}_{\text{n}}(n') \triangleq \text{better}(\text{import}_{\text{n,n'}}(\text{export}_{\text{n',n}}(\text{best}(n'))), \text{best}(n))$$

Where $better_n(p, p')$ is true when path p is preferred over p' according to the ranking function of n.

At any step of the execution, if there is no enabled node, RPVP has reached a converged state. Otherwise a node n is non-deterministically picked among the enabled set (line 9). If the current best path of n is invalid, the best path is set to $\perp$. Among all peers of n that can produce advertisements that can update the best path of n, the neighbors that produce the highest ranking advertisements are selected (line 13). Note that in our model we allow multiple paths to have the same rank, so there may be more than one elements in the set U. Among the updates, one peer n' is non-deterministically selected and the best path of n is updated according to the advertisement of n' (lines 14-16). By the end of line 16, an iteration of RPVP is finished. Note that there are no explicit advertisements propagated; instead nodes are polled for the advertisement that they would generate based on their current best path when needed. The the protocol terminates once a converged state for the target equivalence class is reached. RPVP does not define the semantics of failure or any change to the ranking functions. Any topology changes to be verified are made before the protocol starts its execution and the latest version of the ranking functions are considered.

A natural question is whether or not performing analysis using RPVP is sound and complete with respect to SPVP. Soundness is trivial as each step of RPVP can be simulated using a few steps of SPVP. If we are only concerned about the converged states, RPVP is complete as well:

Theorem 1. *For any converged state reachable from the initial state of the network with a particular set of links L failing at certain steps during the execution of SPVP, there is an* *execution of RPVP with the same import/export filters and ranking functions equal to the latest version of ranking functions in the execution of SPVP, which starts from the initial state in which all links in L have failed before the protocol execution starts, and reaches the same converged state. Particularly, there is a such execution in which at each step, each node takes a best path that does not change during the rest of the execution.*

Proof. The proof can be found in the Appendix. □

Theorem 1 implies that performing model checking using the RPVP model is complete. Note that RPVP does not preserve all the transient states and the divergent behaviors of SPVP. This frees us from checking unnecessary states as we are only interested in the converged states. Yet, even the reduced state space has a significant amount of irrelevant non-determinism. Consequently, we rely on a suite of other domain-specific optimizations (§ 4) to eliminate much of this non-determinism and make model checking practical.

Note that our presentation of RPVP has assumed the that a single best path is picked by each node. This matches our current implementation in that we do not support multipath in all protocols. In a special-case deviation from RPVP, our implementation allows a node running OSPF to maintain multiple best paths, chosen based on multiple neighbors. While we could extend our protocol abstraction to allow multiple best paths at each node, it wouldn't reflect the real-world behavior of BGP which, even when multipath is configured, makes routing decisions based on a single best path. However, such an extension is valid under the constrained filtering and ranking techniques of shortest path routing. Our theorems can be extended to incorporate multipath in such protocols. We omit that to preserve clarity.

3.5 Policies

We primarily target verification of data plane policies over converged states of the network. Similar to VeriFlow [15], we don't implement a special declarative language for policies; a policy is simply an arbitrary function computed over a data plane state and returning a Boolean value. Plankton implements a Policy API where a policy supplies a callback which will be invoked each time the model checker generates a converged state. Plankton gives the callback the newly-computed converged data plane for a particular PEC, as well as the relevant converged states of any other PEC that the current PEC depends on. Plankton checks the callback's return value, and if the policy has failed, it writes a trail file describing the execution path taken to reach the particular converged state.

Our API allows a policy to give additional information to help optimize Plankton's search: *source nodes* and *interesting nodes*. We define two converged data plane states for a PEC to be *equivalent* if their paths from the source nodes have the same length and the same interesting nodes are in the same position on the path. Plankton may suppress checking a converged state if an equivalent one (from the perspective of that policy) has already been checked (§ 4.2 and § 4.3 describe how Plankton does this). If source and interesting nodes are not supplied, Plankton by default assumes that all nodes might be sources and might be interesting.

As an example, consider a waypoint policy: traffic from a set S of sources must pass through firewalls F. The policy specifies sources S, interesting nodes F, and the callback function steps through each path starting at S and fails when it finds a path that does not contain a member of F. As another example, a loop policy can't optimize as aggressively: it has to consider all sources.

In general, this API enables *any policy that is a function of a single PEC's converged data plane state*. We have found it simple to add new policies, currently including: Reachability, Waypointing, Loop Freedom, BlackHole Freedom, Bounded Path Length and Multipath Consistency [1]. We highlight several classes of policies that fall outside this API: (i) Policies that inspect the converged control plane state, as opposed to the data plane: while not yet strictly in the clean API, this information is easily available and we implemented a representative example, Path Consistency (§5), which asserts that the control plane state as well as the data plane paths for a set of devices should be identical in the converged state (similar to Local Equivalence in Minesweeper [1]). (ii) Policies that require multiple PECs, e.g., "packets to two destinations use the same firewall". This would be an easy extension, leveraging Plankton's PEC-dependency mechanism, but we have not performed a performance evaluation. (iii) Policies that inspect dynamic behavior, e.g., "no transient loops prior to convergence", are out of scope just as they are for all current configuration verification tools.

4 Optimizations

Although Plankton's RPVP-based control plane greatly reduces the state space, naive model checking is still not efficient enough to scale to large networks. We address this challenge through optimizations that fall into two major categories — reducing the search space of the model checker, and making the search more efficient.

4.1 Partial Order Reduction

A well-known optimization technique in explicit-state model checking, Partial Order Reduction (POR) involves exploring a set of events only in one order, if the various orderings will result in the same outcome. In general, precisely answering whether the order of execution affects the outcome can be as hard as model checking itself. Model checkers such as SPIN provide conservative heuristics to achieve some reduction. However, in our experiments, this feature did not yield any reduction. We believe this is because our model of the network has only a single process, and SPIN's heuristics are designed to apply only in a multiprocess environment. Even if we could restructure the model to use SPIN's heuristics, we do not expect significant reductions, as evidenced in past work [4]. Instead, we implement POR heuristics, based on our knowledge of the RPVP control plane[†].

4.1.1 Explore consistent executions only

To describe this optimization, we first introduce the notion of a *consistent* execution: For a converged state S and a partial execution of RPVP π , we say that π is consistent with S iff at each step of the execution, a node picks a path that is equal to it its best path in S and never changes it.

Readers may notice that Theorem 1 asserts the existence of a consistent execution leading to each converged state of the network, once any failures have happened. This implies that if the model checker was to explore only executions that are consistent with some converged state, completeness of the search would not be compromised (soundness is not violated since every such execution is a valid execution). Of course, when we start the exploration, we cannot know the exact executions that are consistent with some converged state, and hence need to be checked. So, we conservatively assume that every execution we explore is relevant, and if we get evidence to the contrary (like a device having to change a selected best path), we stop exploring that execution.

4.1.2 Deterministic nodes

Even when there is the possibility of non-deterministic convergence, the "relevant" non-determinism is typically localized. In other words, after each non-deterministic choice, there is a significant amount of *essentially* deterministic behavior before the opportunity for another non-deterministic choice, if any, arises. (We consider it analogous to endgames in chess, but applicable at any point in the protocol execution.) However, this is obscured by "irrelevant" non-determinism – particularly, ordering between node execution that doesn't impact the converged state. Our goal is to prune the irrelevant non-determinism to reduce the search space for Plankton's model checker.

For an enabled node n in state S with a single best update u, we say n is *deterministic* if in all possible converged states reachable from S, n will have the path selected after n processes u. Of course, with the model checker having only partially executed the protocol, it is highly non-obvious which nodes are deterministic! Nevertheless, suppose for a moment we have a way to identify at least *some* deterministic nodes. How could we use this information? At each step of RPVP, after finding the set of enabled nodes, if we can identify at least one deterministic enabled node, we choose *one* of these nodes and instruct SPIN to process its update. (More specifically, we pick one arbitrarily.) This avoids the costly non-deterministic branching caused by having to explore the numerous execution paths where *each one* of the enabled nodes is the one executed next. The following theorem shows this is safe.

[†]Since we wish to check all converged states of the network, it can be argued that any reduction in search space is essentially POR. But here, we are referring optimizations that have a localized scope.

Theorem 2. *Any partial execution of RPVP consistent with a converged state can be extended to a complete execution consistent with that state.*

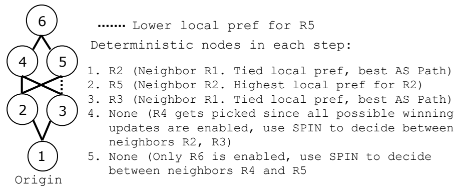
Proof. The proof can be found in the Appendix. □

By definition, choosing *any* deterministic node as the single node to execute next produces a new state that remains consistent with all possible converged states reachable from the prior state. Thus, Theorem 2 implies this deterministic choice does not eliminate any converged states from being reachable, preserving completeness. Note that this optimization does not require the entire network to have one converged state; it can apply at every step of the execution, possibly between non-deterministic choices.

What remains is the key: how can we identify deterministic nodes? Perfect identification is too much to hope for, and we allow our heuristics to return fewer deterministic nodes than actually exist. We build heuristics that are specific to each routing protocol, prioritizing speed and simplicity above handling atypical cases like circular route redistribution.

For OSPF, our detection algorithm runs a network-wide shortest path computation, and picks each node only after all nodes with shorter paths have executed. We cache this computation so it is only run once for a given topology, set of failures, and set of sources.

For BGP, the detection algorithm performs the following computation: For each node which is enabled to update its best path, it checks whether there exists a pending update that would never get replaced, because the update would definitely be preferred over other updates that are enabled now or may be in the future. To check this, we follow the node's BGP decision process, so if the update is tied for most-preferred in one step it moves to the next. For each step of the decision process, the preference calculation is quite conservative. For local pref, it marks an update as the winner if it matches an import filter that explicitly gives it the highest local pref among all import filters. For AS Path, the path length of the current update must be the minimum possible in the topology. For IGP cost, the update must be from the peer with minimum cost. If at any node, any update is found to be a clear winner, the node is picked as a deterministic node, and is allowed to process the update. If no node is found that has a clear winner but there is a node that has ≥ 2 updates tied for the most preferred, then we deterministically pick any one such node and have SPIN non-deterministically choose which of the multiple updates to process. Figure 6 illustrates these scenarios on a BGP network, highlighting one sequence of node selections (out of many possible).



```
Deterministic nodes in each step:

1. R2 (Neighbor R1. Tied local pref, best AS Path)
2. R5 (Neighbor R2. Highest local pref for R2)
3. R3 (Neighbor R1. Tied local pref, best AS Path)
4. None (R4 gets picked since all possible winning
   updates are enabled, use SPIN to decide between
   neighbors R2, R3)
5. None (Only R6 is enabled, use SPIN to decide
   between neighbors R4 and R5
```

Figure 6: Step-by-step choice of deterministic nodes (Each node has a different AS number).

The detection algorithm may fail to detect some deterministic nodes. For instance, suppose node N is deterministic but its import filter from neighbor M sets the highest local pref for updates with a particular community attribute, and M can never assign that attribute. Then the detection algorithm will fail to mark N as deterministic. But successfully identifying *at least one* deterministic node in a step will avoid non-deterministic branching at that step. As long as this happens frequently, the optimization will be helpful.

Even if the decision on a node is ambiguous in a particular state, the system will often make progress to a state where ambiguities can be resolved. In the example above, once M selects a path (and therefore will never change its path as described in § 4.1.1), the detection algorithm no longer needs to account for a possible more-preferred update from it, and may then be able to conclude that N is deterministic.

4.1.3 Decision independence

If node A's actions are independent of any future decisions of node B and vice versa, then the execution ordering between A and B does not matter. We check a sufficient condition for independence: any route advertisements passed between these nodes, in either direction, must pass through a node that has already made its best path decision (and therefore will not transmit any further updates). In this case, we pick a single arbitrary execution order between A and B.

4.1.4 Failure ordering

As stated in § 3.4.2, the model checker performs all topology changes before the protocol starts execution. We also enforce a strict ordering of link failures, reducing branching even further.

4.2 Policy-based Pruning

Policy-based pruning limits protocol execution to those parts of the network that are relevant to the satisfaction/failure of the policy. When a policy defines a set of source nodes (§ 3.5), it indicates that the policy can be checked by analyzing the forwarding from those nodes only. The best example for this is reachability, which is checked from a particular set of starting nodes. When an execution reaches a state where all source nodes have made their best-path decision, Plankton considers the execution, which is assumed to be consistent, to have finished. In the cases where only a single prefix is defined in a PEC, Plankton performs a more aggressive pruning, based on the influence relation. Any device that cannot influence a source node is not allowed to execute. With some additional bookkeeping, the optimization can be extended to cases where multiple prefixes contribute to a PEC, but our current implementation does not support this. The optimization is also not sound when applied to PECs on which other PECs depend. A router that does not speak BGP may not directly influence a source node, but it may influence the routing for the router IP addresses, which in turn may affect the chosen path of the source node. So, the optimization is not applied in such cases.

4.3 Choice of Failures

In addition to the total ordering of failures described in § 4.1.4, Plankton also attempts to reduce the number of failures that are explored, using equivalence partitioning of devices as proposed by Bonsai [2]. Bonsai groups devices in the network into abstract nodes, creating a smaller topology overall for verification. Plankton computes Device Equivalence Classes (DECs) similarly, and defines a Link Equivalence Class (LEC) as the set of links between two DECs. For each link failure, Plankton then explores only one representative from each LEC. When exploring multiple failures, we refine the DECs and LECs after each selection. Note that this optimization limits the choice of failed links, but the verification happens on the original input network. In order to avoid remapping interesting nodes (§ 3.5), they are each assigned to a separate DEC. Since the computed DECs can be different for each PEC, this optimization is done only when there are no cross-PEC dependencies.

4.4 State Hashing

During the exhaustive exploration of the state space, the explicit state model checker needs to track a large number of states simultaneously. A single network state consists of a copy of all the protocol-specific state variables at all the devices. Maintaining millions of copies of these variables is naturally expensive, and in fact, unnecessary. A routing decision at one device doesn't immediately affect the variables at any of the other devices. Plankton leverages this property to the reduce memory footprint, storing routing table entries as 64-bit pointers to the actual entry, with each entry stored once and indexed in a hash table. We believe this optimization can be applied to other variables in the network state too, as long as they are not updated frequently. Picking the right variables to optimize this way, and developing more advanced hash-based schemes, can be explored in the future.

5 Evaluation

We prototyped Plankton including the equivalence class computation, control plane model, policy API and optimizations in 373 lines of Promela and 4870 lines of C++, excluding the SPIN model checker. We experimented with our prototype on Ubuntu 16.04 running on a 3.4 GHz Intel Xeon processor with 32 hardware threads and 188 GB RAM.

We begin our evaluation with simple hand-created topologies incorporating protocol characteristics such as shortest path routing, non-deterministic protocol convergence, redistribution, recursive routing, etc. Among these tests, we incorporated examples of non-deterministic protocol execution from [8], as well as BGP wedgies, which are policy violations which can occur only under some non-deterministic execution paths. In each of these cases, Plankton correctly finds any violations that are present.

Having tested basic correctness, we next evaluate performance and compare to past verifiers. What sets Plankton apart from tools other than Minesweeper is its higher data plane coverage and ability to handle multiple protocols (Figure 1). We therefore compare primarily with Minesweeper but also include ARC in some tests.

We also evaluated Bonsai [2], a preprocessing technique that helps improve the scalability of configuration verification, for specific policies. Bonsai could assist any configuration verifier. We integrated Bonsai with Plankton, and experimentally compare the performance of Bonsai+Minesweeper and Bonsai+Plankton. However, it is important to study the performance of these tools without Bonsai too: Bonsai's network compression cannot be applied if the correctness is to be evaluated under link failures, or if the policy being evaluated is not preserved by Bonsai.

Experiments with synthetic configurations

Our first set of performance tests uses fat trees. We construct fat trees of increasing sizes, with each edge switch originating a prefix into OSPF. Link weights are identical. We check these networks for routing loops. In order to cause violations, we install static routes at the core routers. In our first set of experiments, the static routes match the routes that OSPF would eventually compute, so there are no loops. Then, we change the static routes such that some of the traffic falls into a routing loop. Figure 7(a) illustrates the time and memory consumed, using Plankton running on various numbers of cores, and using Minesweeper. We observed that under default settings, Minesweeper's CPU utilization keeps changing, ranging from 100% to $1,600\%$. In this experiment and all others where we run both Minesweeper and Plankton, the two tools produced the same policy verification results. This serves as an additional correctness check for Plankton. Bonsai is not used, because its currently available implementation does not appear to support loop policies.

As the results show, Plankton scales well with input network size. The speed and memory consumption varies as expected with the degree of parallelism. Even on a single core, Plankton is quicker than Minesweeper for all topologies. For larger networks, Plankton is several orders of magnitude quicker. On the memory front, even on 16 cores, Plankton's footprint is smaller than Minesweeper's.

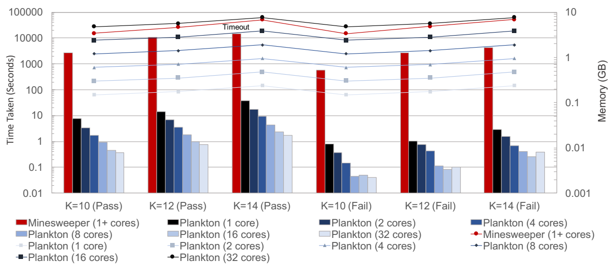
Encouraged by the good performance numbers, we scale up to very large fat trees (Figure 7(b)). Here, Minesweeper doesn't finish even in 4 hours, even with a 500-device network (in the case of passing loop check, even in a 245-device network). So, we did not run Minesweeper on the larger networks. We run Plankton with a single CPU core only, to illustrate its time-memory tradeoff: since the analyses of individual PECs are fully independent and of identical computational effort, running with n cores would reduce the time by $n\times$, and increase memory by $n\times$. For example, in the 2,205-device network, Plankton uses about 170 GB per process. Policies that check a single equivalence class are much cheaper: for example, single-IP reachability finishes in seconds or minutes even on the largest networks (Figure 7(b)).

Next, we test Plankton with a very high degree of non-determinism. We evaluated a data center setting with BGP, which is often employed to provide layer 3 routing down to the rack level in modern data centers [17]. We configure BGP as described in RFC 7938 [17] on fat trees of various sizes. Furthermore, we suppose that the network operator intends traffic to pass through any of a set of acceptable *waypoints* on the aggregation layer switches (e.g., imagine the switches implement a certain monitoring function). We pick a random subset of aggregation switches as the waypoints in each experiment. However, we create a "misconfiguration" that prevents multipath and fails to steer routes through the waypoints[‡]. Thus, in this scenario, whether the selected path passes through a waypoint depends on the order in which updates are received at various nodes, due to age-based tie breaking [16]. We check waypoint policies which state that the path between two edge switches should pass through one of the waypoints. Plankton evaluates various non-deterministic convergence paths in the network, and determines a violating sequence of events. Time and memory both depend somewhat on the chosen set of aggregation switches, but even the worst-case times are less than 2 seconds (Figure 7(c)). We consider this a success of our policy-based pruning optimization: the network has too many converged states to be verified in reasonable time, but many have equivalent results in terms of the policy.

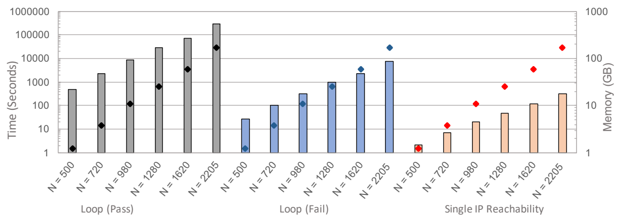
Experiments with semi-synthetic configurations

We use real-world AS topologies and associated OSPF link weights obtained from RocketFuel [24]. We pick a random ingress point that has more than one link incident on it. We verify that with any single link failure, all destination prefixes are reachable from that ingress. Here, Minesweeper's SMT-based search algorithm could be beneficial, due to the large search space created by failures. Nevertheless, Plankton performs consistently better in both time and memory (Figure 7(d)). Both tools find a violation in each case. The time taken by Plankton with 16 and 32 cores are often identical, since a violation is found in the first set of PECs. Note

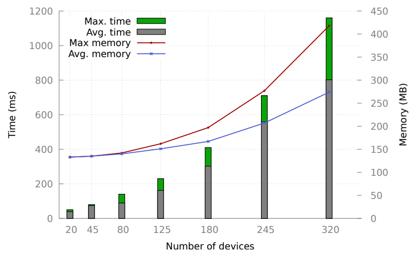
[‡]This setup is convenient for practical reasons, as our current Plankton prototype implementation does not support BGP multipath.



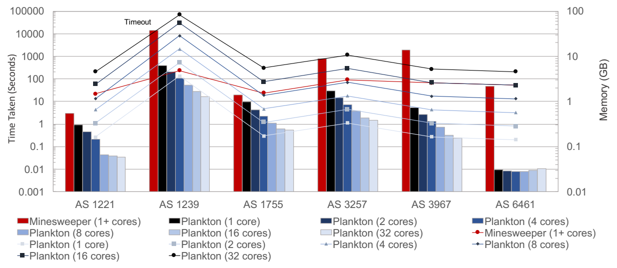
(a) Fat trees with OSPF, loop policy, multi-core



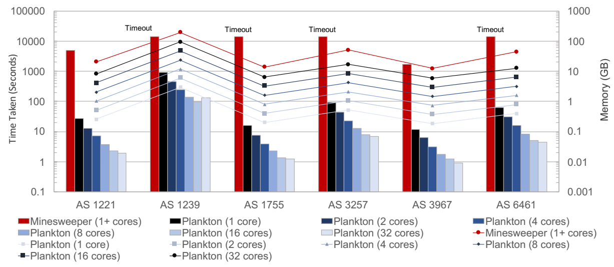
(b) Fat trees with OSPF, multiple policies, 1 core



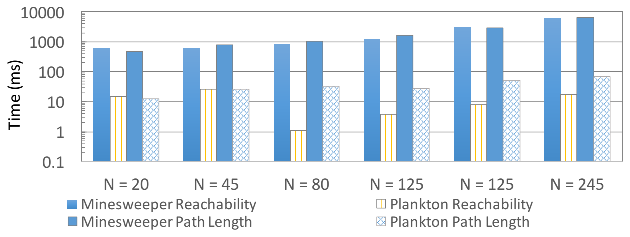
(c) Fat trees with BGP, waypoint policy, 1 core



(d) AS topologies with OSPF and failures, reachability policy, multi-core



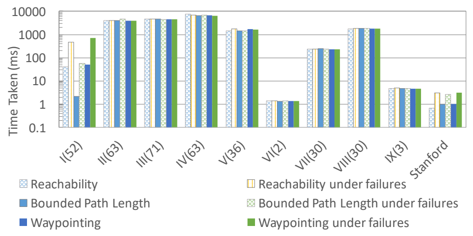
(e) AS topologies with iBGP over OSPF, reachability policy, multi-core



(f) Bonsai-compressed fat trees with OSPF, multiple policies, 8 cores

Network	Links Failed	ARC	Plankton
Fat tree (20 nodes)	0	1.08 s	0.19 s
	≤ 1	1.05 s	0.23 s
	≤ 2	1.00 s	0.31 s
Fat tree (45 nodes)	0	12.17 s	0.49 s
	≤ 1	12.40 s	0.55 s
	≤ 2	12.49 s	2.09 s
Fat tree (80 nodes)	0	300.50 s	0.93 s
	≤ 1	280.00 s	1.98 s
	≤ 2	294.12 s	18.30 s
Fat tree (125 nodes)	0	4847.57 s	1.90 s
	≤ 1	5096.96 s	9.34 s
	≤ 2	4955.95 s	159.37 s
AS 1221 (108 nodes)	0	41.62 s	1.12 s
	≤ 1	40.50 s	2.80 s
	≤ 2	38.83 s	106.57 s
AS 1775 (87 nodes)	0	49.32 s	1.02 s
	≤ 1	48.65 s	2.73 s
	≤ 2	46.52 s	155.28 s

(g) Networks with link failures, all-to-all reachability policy, 8 cores



(h) Real-world configs (number of devices in parentheses), multiple policies, 1 core

Network	Policy	Links Failed	Memory	Time
II	Loop	0	2.37 GB	8.79 s
		≤ 1	2.47 GB	13.62 s
	Multipath Consistency	0	2.37 GB	16.28 s
		≤ 1	2.37 GB	22.01 s
	Path Consistency	0	2.37 GB	15.43 s
		≤ 1	2.37 GB	23.55 s
III	Loop	0	2.36 GB	11.49 s
		≤ 1	2.88 GB	29.81 s
	Multipath Consistency	0	2.37 GB	16.33 s
		≤ 1	2.67 GB	24.86 s
	Path Consistency	0	2.36 GB	15.53 s
		≤ 1	2.88 GB	21.01 s
IV	Loop	0	2.31 GB	12.37 s
		≤ 1	2.40 GB	13.14 s
	Multipath Consistency	0	2.34 GB	16.36 s
		≤ 1	2.37 GB	17.04 s
	Path Consistency	0	2.33 GB	16.33 s
		≤ 1	2.37 GB	17.00 s

(i) Real-world configs, multiple policies, 32 cores

Figure 7: Plankton experiments. Bars and lines/points denote time and memory consumption, respectively.

that in this experiment and the next, we did not use Bonsai, because (i) it cannot be used for checks involving link failures, and (ii) the topology has hardly any symmetry that Bonsai could exploit.

To evaluate our handling of PEC dependencies, we configure iBGP over OSPF on the AS topologies. The iBGP prefixes rely on the underlying OSPF routing process to reach the next hop. We check that packets destined to the iBGP-announced prefixes are correctly delivered. It is worth noting that this test evaluates a feature that, to the best of our knowledge, is provided only by Plankton and Minesweeper. Thanks to the dependency-aware scheduler, Plankton performs multiple orders of magnitude better (Figure 7(e)). This is unsurprising: Minesweeper's approach of duplicating the network forces it to solve a *much* harder problem here, sometimes over $300\times$ larger.

Integration with Bonsai

We integrated Plankton with Bonsai to take advantage of control plane compression when permitted by the specific verification task at hand. We test this integration experimentally by checking Bounded Path Length and Reachability policies on fat trees running OSPF. The symmetric nature of fat trees is key for Bonsai to have a significant impact. We measure the time taken by Plankton and Minesweeper, *after* Bonsai preprocesses the network. Plankton still outperforms Minesweeper by multiple orders of magnitude (Figure 7(f)).

Comparison with ARC

Having evaluated Plankton's performance in comparison with Minesweeper, we move on to comparing the performance of Plankton and ARC. ARC is specifically designed to check shortest-path routing under failures, so we expected the performance to be much better than the more general-purpose Plankton, when checking networks compatible with ARC. We check all-to-all reachability in fat trees and AS topologies running OSPF, under a maximum of 0, 1 and 2 link failures. Similar to Minesweeper, ARC's CPU utilization ranges from 100% to 600% under default settings. We allocate 8 cores to Plankton. Plankton is multiple orders of magnitude faster in most cases (Figure 7(g)).[§] This is genuinely surprising; one reason that *may* explain the observation is that ARC always computes separate models for each source-destination pair, whereas Plankton computes them based only the destination, when verifying destination address routing. Nevertheless, we do not believe that there is a fundamental limitation in ARC's design that would prevent it from outperforming Plankton on the networks that can be checked by either tool. Interestingly, while ARC's resiliency-focused algorithm doesn't scale as easily as Plankton for larger networks, its performance actually sometimes slightly *improves* when the number of failures to be checked increases. Plankton on the other hand scales poorly when checking increasing levels of resiliency. We do not find this concerning, since most interesting checks in the real world involve only a small number of failures. When we performed these experiments with Minesweeper, no check involving 2 failures ran to completion except the smallest fat tree.

Testing with real configurations

We used Plankton to verify 10 different real-world configurations from 3 different organizations, including the publicly available Stanford dataset. We first check reachability, waypointing and bounded path length policies on these networks, with and without failures. All except one of these networks use some form of recursive routing, such as indirect static

[§]Our numbers for ARC are similar to those reported by its authors for similar sized networks, so we believe we have not misconfigured ARC.

Experiment	Optimizations	Time	Memory
Ring, OSPF, 4 nodes, 1 failure	All	343 μs	137.43 MB
	None	1.56 ms	137.39 MB
Ring, OSPF, 8 nodes, 1 Failure	All	623 μs	143.22 MB
	None	0.13 s	137.04 MB
Ring, OSPF, 16 nodes, 1 Failure	All	2.44 ms	137.89 MB
	None	266.48 s	7615.57 MB
Fat tree, OSPF, 20 nodes	All	464 μs	551.73 MB
	None	> 5 min	> 8983.55 MB
Fat tree, OSPF, 245 nodes	All	4.297 s	1908 MB
	All but link failure opt.	64.97 s	72862 MB
AS 1221 iBGP	All	27.54 s	254.22 MB
	All but deterministic node opt.	25.43 s	254.34 MB
Fat tree, BGP, 20 nodes	All	46 ms	137 MB
	All but deterministic node opt.	> 5 min	> 6144 MB
	All but policy-based pruning	> 5 min	> 6144 MB

Figure 8: Experiments with optimizations disabled/limited

routes or iBGP. We feel that this highlights the significance of Plankton's and Minesweeper's support for such configurations. Moreover, the PEC dependency graph for these networks did not have any strongly connected components larger than a single PEC, which matches yet another of our expectations. Interestingly, we did find that the PEC dependency graph had *self loops*, with a static route pointing to a next hop IP within the prefix being matched. It is also noteworthy that in these experiments, the only non-determinism was in the choice of links that failed, which substantiates our argument that network configurations in the real world are largely deterministic. Figure 7(h) illustrates the results, which indicate that Plankton can handle the complexity of real-world configuration verification.

In our next experiment with real world configs, we identify three networks where Loop, Multipath Consistency and Path Consistency policies are meaningful and non-trivial to check. We check these policies with and without link failures. Figure 7(i) illustrates the results of this experiment. The results indicate that the breadth of Plankton's policies scale well on real world networks. The Batfish parser, which is used by Minesweeper, was incompatible with the configurations, so we could not check these configs on Minesweeper (checking the Stanford dataset failed *after* parsing). However, the numbers we observe are significantly better than those reported for Minesweeper on similar-sized networks, for similar policies.

Optimization Cost/Effectiveness

To determine the effectiveness of Plankton's optimizations, we perform experiments with some optimizations disabled or limited. Figure 8 illustrates the results from these experiments. When all optimizations are turned off, naive model checking fails to scale beyond the most trivial of networks. The optimizations reduce the state space by $4.95\times$ in smaller networks and by as much $24{,}968\times$ in larger ones.

To evaluate device-equivalence based optimizations in picking failed links, we perform loop check on fat trees running OSPF under single link failure with the optimization turned off. We observed a $15\times$ reduction in speed, and a $38\times$ increase in memory overhead, indicating the effectiveness of the optimization in networks with high symmetry.

Experiment	No Bitstate Hashing	Bitstate Hashing
180 Node BGP DC Waypoint (Worst Case)	202 MB	67 MB
320 Node BGP DC Waypoint (Worst Case)	428 MB	215 MB
AS 1239 Fault Tolerance (2 cores)	7.33 GB	4.52 GB
AS 1221 Fault Tolerance (1 core)	163.53 MB	60 MB

Figure 9: The effect of bitstate hashing on memory usage

In the next set of experiments, we measure the impact of our partial order reduction technique of prioritizing deterministic nodes (§ 4.1.2). We first try the iBGP reachability experiment with the AS 1221 topology, with the detection of deterministic nodes in BGP disabled. We notice that in this case the decision independence partial order reduction produces reductions identical to the disabled optimization, keeping the overall performance unaffected. In fact, the the time improves by a small percentage, since there is no detection algorithm that runs at every step. We see similar results when we disable the optimization on the edge switches in our BGP data center example. However, this does not mean that the deterministic node detection can be discarded — in the BGP data center example, when the optimization is disabled altogether, the performance falls dramatically. The next optimization that we study is policy-based pruning. On the BGP data center example, we attempt to check a waypoint policy, with policy-based optimizations turned off. The check times out, since it is forced to generate every converged data plane, not just the ones relevant to the policy.

SPIN provides a built-in optimization called *bitstate hashing* that uses a Bloom filter to keep track of explored states, rather than storing them explicitly. This can cause some false negatives due to reduced coverage of execution paths. We find that bit state hashing provides significant reduction in memory in a variety of our test cases (Figure 9). According to SPIN's statistics our coverage would be over 99.9%. Nevertheless, we have not turned on bitstate hashing in our other experiments in favor of full correctness.

6 Limitations

Some of the limitations of Plankton, such as the lack of support for BGP multipath and limited support for route aggregation, have been mentioned in previous sections. As discussed in § 3.2, Plankton may also produce false positives when checking networks with cross-PEC dependencies, because it expects that every converged state of a PEC may coexist with every converged state of other PECs that depend on it. However, such false positives are unlikely to happen in practice, since real-world cases of cross-PEC dependencies (such as iBGP) usually involve only a single converged state for the recursive PECs. Our current implementation of Plankton assumes full visibility of the system to be verified, and that any dynamics will originate from inside the system. So, influences such as external advertisements need to be modeled using stubs that denote entities which originate them. Plankton's technique is not suited for detecting issues in vendor-specific protocol implementations, a limitation that all existing formal configuration analysis tools share. As with most formal verification tools, one needs to assume that Plankton itself is correct, both in terms of the theoretical foundations as well as the implementation. Correct-by-construction program synthesis could help in this regard.

7 Related Work

Data plane verification: The earlier offline network verification techniques [19, 13] have evolved into more efficient and real-time ones (e.g., [15, 12, 10, 26, 3, 11]), including richer data plane models (e.g., [21, 14]). These techniques however, cannot verify configurations prior to deployment.

Configuration verification: We discussed the state of the art of configuration verification in § 2, and how Plankton improves upon the various tools in existence. CrystalNet [18] emulates actual device VMs, and its results could be fed to a data plane verifier. However, this would not verify non-deterministic control plane dynamics. Simultaneously improving the fidelity of configuration verifiers in *both* dimensions (capturing dynamics as in Plankton and implementation-specific behavior as in CrystalNet) appears to be a difficult open problem.

Optimizing network verification: Libra [27] is a divide-and-conquer data plane verifier, which is related to our equivalence class-based partitioning of possible packets. The use of symmetry to scale verification has been studied in the data plane [22] and control plane (Bonsai [2]). We have discussed how Plankton uses ideas similar to Bonsai, as well as integrates with Bonsai itself.

Model checking in the networking domain: Past approaches that used model checking in the networking domain have focused almost exclusively on the network software itself, either as SDN controllers, or protocol implementations [4, 23, 20]. Plankton uses model checking not to verify software, but to verify configurations.

8 Conclusion and Future Work

We described Plankton, a formal network configuration verification tool that combines equivalence partitioning of the header space with explicit state model checking of protocol execution. Thanks to pragmatic optimizations such as partial order reduction and state hashing, Plankton produces significant performance gains over the state of the art in configuration verification. Improvements such as checking transient states, incorporating real software, partial order reduction heuristics that guarantee reduction, etc. are interesting avenues of future work.

Acknowledgements: We thank the anonymous reviewers and our shepherd Cole Schlesinger for their comments. We also thank Ratul Mahajan and Ryan Beckett for their help in integrating Bonsai with Plankton. This work was supported by NSF grants CNS-1513906 and CCF-1421575.

References

[1] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 155–168.

[2] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 476–489.

[3] Bjørner, N., Juniwal, G., Mahajan, R., Seshia, S. A., and Varghese, G. ddnf: An efficient data structure for header spaces. In *Haifa Verification Conference* (2016), Springer, pp. 49–64.

[4] Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. A NICE way to test openflow applications. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 127–140.

[5] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., and Varghese, G. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), pp. 217–232.

[6] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. A general approach to network configuration analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 469–483.

[7] Gember-Jacobson, A., Viswanathan, R., Akella, A., and Mahajan, R. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), SIGCOMM '16.

[8] Griffin, T. G., Shepherd, F. B., and Wilfong, G. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw. 10*, 2 (Apr. 2002), 232–243.

[9] Holzmann, G. J. The model checker spin. *IEEE Trans. Softw. Eng. 23*, 5 (May 1997), 279–295.

[10] Horn, A., Kheradmand, A., and Prasad, M. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 735–749.

[11] Horn, A., Kheradmand, A., and Prasad, M. R. A precise and expressive lattice-theoretical framework for efficient network verification. In *2019 27st IEEE International Conference on Network Protocols (ICNP)* (2019), IEEE.

[12] Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S. Real time network policy checking using header space analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013).

[13] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 113–126.

[14] Kheradmand, A., and Rosu, G. P4K: a formal semantics of P4 and applications. *CoRR abs/1804.01468* (2018).

[15] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying Network-wide Invariants in Real Time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), NSDI'13, USENIX Association, pp. 15–28.

[16] Lapukhov, P. Equal-Cost Multipath Considerations for BGP. Internet-Draft draft-lapukhov-bgp-ecmp-considerations-00, Internet Engineering Task Force, Oct. 2016. Work in Progress.

[17] Lapukhov, P., Premji, A., and Mitchell, J. Use of BGP for Routing in Large-Scale Data Centers. RFC 7938 (Informational), Aug. 2016.

[18] Liu, H. H., Zhu, Y., Padhye, J., Cao, J., Tallapragada, S., Lopes, N. P., Rybalchenko, A., Lu, G., and Yuan, L. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 599–613.

[19] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, pp. 290–301.

[20] Musuvathi, M., and Engler, D. R. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1* (Berkeley, CA, USA, 2004), NSDI'04, USENIX Association, pp. 12–12.

[21] Panda, A., Lahav, O., Argyraki, K., Sagiv, M., and Shenker, S. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 699–718.

[22] Plotkin, G. D., Bjørner, N., Lopes, N. P., Rybalchenko, A., and Varghese, G. Scaling network verification using symmetry and surgery. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2016), POPL '16, ACM, pp. 69–83.

[23] Sethi, D., Narayana, S., and Malik, S. Abstractions for model checking SDN controllers. In *2013 Formal Methods in Computer-Aided Design* (oct 2013), IEEE.

[24] Spring, N., Mahajan, R., Wetherall, D., and Anderson, T. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw. 12*, 1 (Feb. 2004), 2–16.

[25] Weitz, K., Woos, D., Torlak, E., Ernst, M. D., Krishnamurthy, A., and Tatlock, Z. Scalable verification of border gateway protocol configurations with an smt solver. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2016), OOPSLA 2016, ACM.

[26] Yang, H., and Lam, S. S. Real-time verification of network properties using atomic predicates. In *2013 21st IEEE International Conference on Network Protocols (ICNP)* (2013), IEEE, pp. 1–11.

[27] Zeng, H., Zhang, S., Ye, F., Jeyakumar, V., Ju, M., Liu, J., McKeown, N., and Vahdat, A. Libra: Divide and conquer to verify forwarding tables in huge networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 87–99.

A Extended SPVP

In extended SPVP, for each node n, and for each peer $n' \in \text{peers}(n)$, $\text{rib-in}_n(n')$ keeps track of the most recent advertisement of n' to n. In addition, $\text{best-path}(n)$ keeps the best path that n has to one of the *multiple* origins. Peers are connected using reliable FIFO message buffers to exchange advertisements. Each advertisement consists of a path from the advertising node to an origin. In each step (which we assume is performed atomically) a node n takes an advertisement

p from the buffer connected to peer n', and applies an import filter on it ($\mathrm{import}_{n,n'}(p)$). n then updates $\text{rib-in}_n(n')$ with the new imported advertisement. In our extension, we assume that each node has a ranking function λ that provides a *partial order* over the paths acceptable by the node. n then proceeds by updating its best-path to the highest ranking path in rib-in_n. If the best path in rib-in_n have the same rank as the current best path and that path is still valid, $\text{best-path}(n)$ will not change. If the best path is updated, n advertises the path to its peers. For each peer n', n applies the export filter on the path ($\mathrm{export}_{n,n'}(\text{best-path}(n))$) before sending the advertisement.

The import filter, the export filter, and the ranking functions are abstract notions that will be inferred from the configuration of the node. We make reasonable assumptions about these notions (Appendix B). Attributes such as local pref, IGP cost, etc. are accounted for in the the ranking function and the import/export filters.

If a session between two peers fails, the messages in the buffer are lost and the buffer cannot be used anymore. We assume that when this happens, each peer gets $\perp$ as the advertised path. Additionally, to be able to model iBGP, in extended SPVP we allow the ranking function of any node n to change at any time during the execution of the protocol. This is to model cases in which for example a link failure causes IGP costs to change. In such cases we assume that n receives a special message to recompute its best-path according to the new ranking function.

The state of network at each point in time consists of the values of best-path, rib-in, and the contents of the message buffers. In the initial state S_0, the best path of the origins is ε and the best path of the rest of the nodes are $\perp$ which indicates that the node has no path. Also for any $n, n' \in V$, $\text{rib-in}_n(n')$ is $\perp$. We assume that initially the origins put their advertisements in the message buffer to their peers, but the rest of the buffers are empty.

An (partial) execution of SPVP is a sequence of states π which starts from S_0 and each state is reachable by a single atomic step of SPVP on the state before it. A converged state in SPVP, is a state in which all buffers are empty. A complete execution is an execution that ends in a converged state. It is well known that there are configurations which can make SPVP diverge in some or all execution paths. However, our goal is to only to check the forwarding behavior in the converged states, through explicit-state model checking. So, we define a much simpler model that can be used, without compromising the soundness or completeness of the analysis (compared to SPVP).

B Assumptions

We make the following assumptions in in our theoretical model.

- Both import/export filters return $\perp$ if the filter rejects the advertisement according to the configuration.
- All import filters reject paths that cause forwarding loops. They also do not alter the path (unless the path is rejected).
- All export filters for each node n not rejecting an advertisement, will append n at the end of the advertised path. No other modification is made to the path.
- Path $\perp$ has the lowest ranking in all ranking functions.
- The import/export filters never change during the execution of the protocol. Note that we do not make such assumption for the ranking functions.

Note that these are reasonable assumptions with respect to how real world protocols (especially BGP) work.

C Proof of Theorems

Proof of Theorem 1. For a complete execution $\pi = S_0, S_1, ..., S_c$ of SPVP and a state S_i in that execution, for any node n, we say that n is converged in S_i for execution π iff n has already picked the path it has in the converged state (S_c) and does not change it:

$$\mathrm{converged}_\pi(n, S_i) \triangleq \\ \forall j. i \le j \le c : \text{best-path}_{S_j}(n) = \text{best-path}_{S_c}(n)$$

It it clear that when a node converges in an state, it remains converged (according to the definition above).

Lemma 1. *In any complete execution $\pi = S_0, S_1, ..., S_c$ of SPVP, for any sate S_i, for any two nodes n and n', if $\exists l : n' = \text{best-path}_{S_i}(n)[l]$ [¶] , and $\text{best-path}_{S_i}(n') \neq \text{best-path}_{S_i}(n)[l :]$, there is a j ($i < j \le c$) such that $\text{best-path}_{S_j}(n) \neq \text{best-path}_{S_i}(n)$.*

Proof. This can be shown by a simple induction on the length of the prefix of the best path of n from n to up to n' (l). If $l = 1$ (i.e the two nodes are directly connected) then either n' will advertise its path to n and n and will change its path or the link between n and n' fails in which case n will receive an advertisement with $\perp$ as the path (Section 3.4.1), which causes n to change its path. Note that the argument holds even if the ranking function of n or n' changes. If $l > 1$, assuming the claim holds for lengths less than l, for $n'' = \text{best-path}_{S_i}(n)[0]$, either $\text{best-path}_{S_i}(n'') \neq \text{best-path}_{S_i}(n)[1 :]$ in which case due to induction hypothesis the claim holds, or $\text{best-path}_{S_i}(n'') = \text{best-path}_{S_i}(n)[1 :]$ in which case we note that $n' = \text{best-path}_{S_i}(n'')[l-1]$, and since the length of the path n'' to n' is less than l, by induction hypothesis we know that eventually (i.e for a $j > i$) $\text{best-path}_{S_j}(n'') \neq \text{best-path}_{S_j}(n)[1 :]$ and by induction hypothesis this will lead to a change in the best path of n.

[¶]For a path $P = p_0, p_1, ..., p_n$, we denote p_i by $P[i]$ and $p_i, p_{i+1}, ..., p_n$ by $P[i :]$.

□

Corollary 1. *For any complete execution π of SPVP , for any node n, any node along the best path of n in the converged state converges before n.*

Now consider a complete execution $\pi = S_0, S_1, ..., S_c$ of SPVP. We will construct a complete execution of RPVP with $|N|$ steps (where N is set of all nodes) resulting in the same converged state as S_c. We start with a topology in which all the links that have failed during the execution of SPVP are already failed. For any node n, we define $C_\pi(n) = min\{i | \text{converged}_\pi(n, S_i)\}$. Consider the sequence $n_1, n_2, ..., n_{|N|}$ of all nodes sorted in the increasing order of C_π. Now consider the execution of RPVP $\pi' = S'_0, S'_1, ..., S'_{|N|}$ which starts from the initial state of RPVP and in each state S'_i, (a) either node n_{i+1} is the picked enabled node and the node $p_i = \text{best-path}_{S_c}(n_{i+1})[0]$ is the picked best peer, or (b) in case $p_i = \perp$, nothing happens and $S'_{i+1} = S'_i$.

First, note that (modulo the repeated states in case *b*), π' is a valid execution of of RPVP: at each state S'_i (in case *a*), n_{i+1} is indeed an enabled node since its best path at that state is $\perp$ and according to corollary 1, p_i has already picked its path: Also p_i will be in the set of best peers of n_{i+1} (line 13 in RPVP). Assume this is not the case, i.e there exists another peer p' that can advertise a better path. This means that in S_c of SPVP, p' can send an advertisement that is better (according to the version of ranking functions in S_c) than the converged path of n_{i+1}. This contradicts the fact that S_c is a converged path.

Second, note that $S'_{|N|}$ is a converged state for RPVP, because otherwise, using similar reasoning as above, S_c can not be converged. Also it is easy to see that $\text{best-path}_{S'_{|N|}} = \text{best-path}_{S_c}$

Finally, note that in π', once a node changes its best path from $\perp$, it does not change its best path again.

□

Proof of Theorem 2. We begin by making two observations about RPVP that are key to the proof:

- RPVP for a prefix can never converge to a state having looping paths.
- If a node u adopts the best path of a neighbor v, v will be next hop of u.

Consider any converged state S. The theorem states that any partial execution that is consistent with S can be extended to a full execution that leads to S. We prove the theorem by induction on the length of the longest best path in S.

Base case: If in a network a converged state exists where the best path at each node is of length 0, that means that each node is either an origin or doesn't have a best path for the prefix. Since any execution apart from the empty execution (where no protocol event happens) is not consistent with this state, the theorem holds.

Induction hypothesis: If a converged state exists in a network such that all best paths are of length k or less, then any partial execution that is consistent with the converged state can be extended to a full execution that reaches the converged state.

Induction step: Consider a network with a converged state S such that the longest best path is of length $k + 1$. We first divide the nodes in the network into two classes — N, which are the nodes with best paths of length k or less, and N', which are nodes with best paths of length $k + 1$. Consider a partial execution π that is consistent with S. We identify two possibilities for π:

Case 1: Every node that has executed in π falls into N. In this case, we define a smaller network which is the subgraph of the original network, induced by the nodes in N. In this network, the path selections made in S will constitute a converged state. This is because in the original network, in the state S, the nodes in N are not enabled to make state changes. So, we can extend π such that we get an execution π' where nodes in N match the path selections in S. Now, we further extend π' with steps where each node in N' reads the best path from the node that is its nexthop in S and updates its best path. When every node in N has done this, the overall system state will reach S.

Case 2: At least one node in N' has executed in π. In this case, we observe that since π is consistent with S, by the definition of a consistent execution, no node in the network has read the state of any node in N'. So, we can construct an execution π' which has the same steps as π, except that any step taken by a node in N' is skipped. As in the previous case, π' can be extended to reach a converged state in the subgraph induced by N. We extend π, first by using the steps that extend π', and if necessary, taking additional steps at nodes from N' to reach S. □

Config2Spec: Mining Network Specifications from Network Configurations

Rüdiger Birkner[1] Dana Drachsler-Cohen[2*] Laurent Vanbever[1] Martin Vechev[1]

[1]*ETH Zürich* [2]*Technion*

Abstract

Network verification and configuration synthesis are promising approaches to make networks more reliable and secure by enforcing a set of policies. However, these approaches require a formal and precise description of the intended network behavior, imposing a major barrier to their adoption: network operators are not only reluctant to write formal specifications, but often do not even know what these specifications are.

We present *Config2Spec*, a system that automatically synthesizes a formal specification (a set of policies) of a network given its configuration and a failure model (e.g., up to two link failures). A key technical challenge is to design a synthesis algorithm which can efficiently explore the large space of possible policies. To address this challenge, *Config2Spec* relies on a careful combination of two well-known methods: data plane analysis and control plane verification.

Experimental results show that *Config2Spec* scales to mining specifications of large networks (>150 routers).

1 Introduction

Consider the task of a network operator who—tired of human-induced network downtimes—decides to rely on formal methods to verify her network-wide configurations [4,14,22,30] or to synthesize them automatically [5,9,10,28,29]. The operator quickly realizes that both verifiers and synthesizers require a specification of the correct intended network-wide behavior. A few generic requirements quickly come to mind: surely she wants her network to ensure reachability. At the same time, she realizes that her network does *way* more than just ensuring reachability. Among others, it needs to enforce load balancing for popular destinations, provide isolation between customers, drop traffic for suspicious prefixes, and reroute business traffic via predefined waypoints—all these under failures and over hundreds of devices. Writing the precise specification seems daunting, especially as most of it has been homegrown over years, by a team of network engineers (some of which do not even work there anymore).

This situation illustrates the difficulty of writing network specifications. Akin to software specifications, formal specifications are hard to write (as hard as writing the program in the first place [20]), debug, and modify [2,21]. Yet, without easier ways to provide network specifications, network verification and synthesis are unlikely to get widely deployed.

Config2Spec We introduce *Config2Spec*, a system that automatically mines a network's specification from its configurations and a failure model (e.g., up to *k* failures). *Config2Spec* is precise: it returns *all* policies that hold under the failure model (no false negatives) and *only* those (no false positives).

Challenges Mining precise network specifications is challenging as it involves exploring two exponential search spaces: *(i)* the space of all possible policies, and *(ii)* the space of all possible network-wide forwarding states. The challenge stems from the fact that individually exploring each of the search spaces can be prohibitive: a search for the true policies is hard since they are a small fraction of the policy space, while a search for the violated policies is hard since these require witnesses (data planes), which are often sparse.

Insights *Config2Spec* addresses the above challenges by combining the strengths of data plane analysis and control plane verification. Data plane analysis enables us to compute the set of policies that hold for a single data plane, thereby providing an efficient way of *pruning* policies. On the other hand, control plane verification is an efficient way of *validating* that a single policy holds for all the data planes. *Config2Spec* combines the two approaches to prune the large space of policies through sampling and data plane analysis and then, to avoid the need of exploring all data planes, validating the remaining policies with control plane verification. The key insight is to dynamically identify the approach providing for better progress. We design predictors which rely on past iterations and the failure model to switch between the two approaches.

[*]Work done while at ETH Zürich.

Scalability While this approach scales, we identify three domain-specific techniques to improve it even further. First, to better utilize the pruning through data plane analysis, we design a *policy-aware* sampler of data planes. We experimentally show that our approach outperforms a random sampler: with typically fewer samples, it leads to pruning substantially more policies. Second, to reduce the number of queries posed to the verifier, we group queries to the control plane verifier. Third, we analyze the network topology to prune policies that are physically not feasible due to poor connectivity of the routers. For large networks and permissive failure models, this technique makes the difference between *Config2Spec* completing in few hours instead of days.

System We implemented *Config2Spec*, which leverages two state-of-the-art data plane analysis and control plane verification tools, Batfish [13] and Minesweeper [4]. As the implementation relies on these two tools, it is tied to configurations and features supported by them. The approach itself, is not limited to any specific type of configuration.

Config2Spec provides a substantial improvement over baselines that use each of the above tools in isolation (up to 8.3x against the best baseline). Further, *Config2Spec* often mines a precise network specification within an hour, and for large networks (> 150 routers) within 2.7 hours (for OSPF configurations) or 13.7 hours (for BGP configurations). We also illustrate that *Config2Spec* can handle real network configurations by running it successfully on Internet2's configurations.

Contributions Our main contributions are:

- A novel approach to automatically mine the specification of a network by leveraging both data plane analysis and control plane verification (§3).
- A dynamic predictor to decide which approach provides for better progress (§4).
- A policy-aware sampler to find data planes that are likely to prune more policies (§5).
- Policy grouping and topology-based trimming to reduce the number of queries posed to the verifier (§6, §7).
- An end-to-end implementation and an extensive evaluation across different topologies and baselines, showing that *Config2Spec* scales to large networks and significantly outperforms possible baselines (§8).

Novelty Several previous works [6, 7, 31] have looked into mining a network's specification by observing the content of the data plane. All of these works are limited to reachability policies and unlike *Config2Spec*, they either approximate the specification or do not consider the impact of failures on the specification. Concretely, they only produce the network's policies which hold when all links and routers are up. In contrast, *Config2Spec* is able to mine precise network specifications for a given failure model.

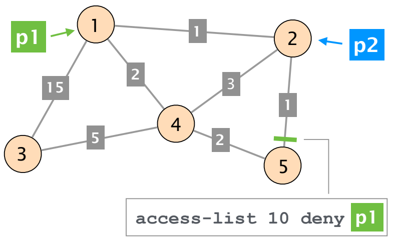


Figure 1: An OSPF network with five routers and two destinations. An ACL at router `5` blocks traffic destined to prefix `p1`, attached to router `1`.

Usefulness In general, the network's specification can be used for many different applications, such as configuration synthesis/verification and network management (e.g., analyzing the effects of configuration changes). Further, having the specification at hand allows network operators to check whether the policies they intend to enforce are indeed enforced.

In multiple discussions, network operators confirmed that a tool like *Config2Spec* is indeed useful. One operator mentioned that the adoption of a new monitoring tool fell through because it required the network's specification to detect flawed configuration changes. Another operator mentioned that having the network's specification at hand would greatly help them better understand their configurations that accumulated over years. Especially, since short-term fixes to problems that need immediate attention (e.g., congestion and hardware problems) are often forgotten and persist even long after the responsible engineer left the company. In addition, the specification can be used to streamline network's configuration by refactoring it, while keeping the same specification.

2 Motivation and Problem Definition

Obtaining a specification for how a network behaves can be useful in a variety of scenarios beyond network verification and synthesis, including helping the operator identify unexpected behaviors and inconsistencies, as well as enabling a smoother transition to (updated) configurations upon new requirements. To define the problem of mining specifications, we rely on two concepts: a *network specification*, composed of a set of *policies*, and a *failure model*, specifying under which failures the network specification should hold. We next define these concepts and illustrate them on a running example. Then, we introduce the network specification mining problem and discuss several baseline approaches together with their shortcomings, thus motivating our solution.

Running example Throughout the paper, we refer to the example shown in Fig. 1. Here, we have a network that consists of five routers and seven links. There are two host networks, `p1` and `p2`, attached to routers `1` and `2`. All routers

Policy	Meaning
`reachability(r, p)`	Traffic from `r` can reach `p`.
`isolation(r, p)`	Traffic from `r` is isolated from `p`.
`waypoint(r, w, p)`	Traffic from `r` to `p` passes through `w`.
`loadbalancing(r, p)`	Traffic from `r` to `p` is load balanced on at least two paths.

Table 1: Network policies (`r` and `w` are routers, `p` is a prefix).

are in the same OSPF area and the OSPF weights are depicted on the links. An IP access control list (ACL) on the interface from router `5` to `2` drops all packets destined to prefix `p1`.

Failure models A failure model consists of a *symbolic environment* and a number k. The symbolic environment defines which links are up or down, and which links may fail. Technically, a symbolic environment is a partition of the network links L into three subsets L_{up}, L_{down}, and $L_{symbolic}$ (i.e., given L_{up} and L_{down}, we can derive $L_{symbolic} = L \setminus (L_{up} \cup L_{down})$). The number k is a bound on the total number of links which can be simultaneously down. A *concrete environment* is a partition of the network links L into two subsets L_{up} and L_{down}. Namely, all links are fixed to a concrete state: up or down. We say that a failure model with a symbolic environment L^{SE}_{up}, L^{SE}_{down}, $L^{SE}_{symbolic}$ and a bound k, captures a concrete environment with L^{CE}_{up} and L^{CE}_{down} if $L^{SE}_{up} \subseteq L^{CE}_{up}$, $L^{SE}_{down} \subseteq L^{CE}_{down}$, and $|L^{CE}_{down}| \leq k$. Intuitively, a failure model captures all concrete environments for which the links in L^{SE}_{up} are up, the links in L^{SE}_{down} are down, and there are at most k links which are down.

For example, a failure model for our running example is $L_{symbolic} = L$ (i.e., L_{up} and L_{down} are the empty sets) and $k = 1$. This model describes any concrete environment with at most one link failure. There are eight concrete environments which meet this failure model: one where no link is down, and seven in which each of the links fails once. Another failure model is $L_{up} = \{$`2-4`$\}$, $L_{down} = \{$`2-5`$\}$, $L_{symbolic} = L \setminus (L_{up} \cup L_{down})$, and $k = 2$. This model describes any concrete environment whose link between routers `2` and `4` is up, the link between `2` and `5` is down, and the rest may be up or down. Since $k = 2$, another failed link is allowed in addition to `2-5`. There are six concrete environments that meet this failure model.

Network specification and policies A *network specification* consists of a set of policies. A *policy* captures a specific behavior in the network (e.g., reachability of two routers). It is modeled with a predicate (a constraint) which, given a concrete environment, evaluates to true if the policy holds for that concrete environment, and false otherwise. For our running example, the `reachability(5, p2)` policy evaluates to true for the concrete environment in which all links are up, and to false for the concrete environment where all links are down. We say a policy holds for a failure model if it holds for all concrete environments captured by the failure model.

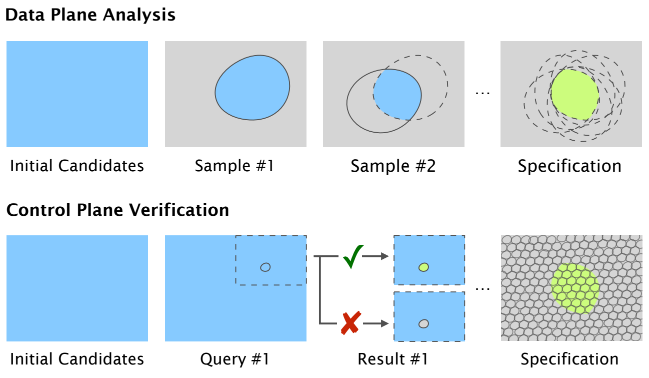


Figure 2: Illustration of the baseline approaches.

For example, the policy `reachability(5, p2)` holds for the failure model $L_{symbolic} = L$ and $k = 1$, but not for $k = 3$.

In our work, we focus on reachability, isolation, waypoint, and load balancing policies (summarized in Table 1). The reachability, isolation, and load balancing policies are defined as predicates over a router `r` and a subnet in the network `p`. These evaluate to true if, for the given concrete environment, traffic from router `r` can reach the prefix `p`, is isolated from `p`, or load balanced on at least two paths to `p`, respectively. The waypoint policy is defined over two routers `r` and `w`, and evaluates to true if, for the given concrete environment, traffic from `r` destined to prefix `p` passes through `w`. We note that our approach is extensible to any policy that is defined over the forwarding state (e.g., equal length paths).

Problem definition We now define the problem of mining a network specification:

Given a network configuration and a failure model, mine the network specification, i.e., the set of all policies which hold under the failure model.

For our running example and the failure model $L_{symbolic} = L$ and $k = 1$ (modeling up to one link failure), the network specification consists of the following policies:
`reachability(1, p1)`, `reachability(1, p2)`,
`reachability(2, p1)`, `reachability(2, p2)`,
`reachability(3, p1)`, `reachability(3, p2)`,
`reachability(4, p1)`, `reachability(4, p2)`,
`reachability(5, p2)`, `loadbalancing(4, p2)`.

Baseline solutions To address the above problem, one may consider two baseline approaches: *(i)* data plane analysis and *(ii)* control plane verification.

Data plane analysis Data plane analysis tools (e.g., [13, 17, 18]) enable reasoning of policies that hold for a certain concrete environment. Today, such tools are scalable enough to reason about all of our considered policies within seconds or minutes (mostly depending on the size of the network). Thus, one could use such tools to mine a specification by iterating over all concrete environments captured by the failure model, computing a data plane for each (from the configuration), and analyzing them to infer the set of policies which hold for each concrete environment. The solution is then the intersection of all obtained policy sets. Fig. 2 (top) visualizes this approach. Initially, every policy is a candidate which can be part of the network specification (blue area). With every sampled data plane, the set of policies that hold for it are computed (shown in circle). These are then intersected with the policies of the previous samples (dashed circles). At the end, the remaining candidate policies are those that hold for all samples, and thus form the network specification (green area). Unfortunately, for large topologies or failure models with many concrete environments, this approach does not scale (see §8.2).

Control plane verification Control plane verification tools (e.g., [4]) enable checking individual policies for a given failure model. Technically, this can be accomplished by symbolically encoding the network, its configuration, the failure model and a policy into a formula, and then checking the satisfiability of this formula. Fig. 2 (bottom) visualizes this approach. Initially, all policies are part of the set of candidates of the specification. At every step, one policy (circle) is picked and posed as a query to the verifier. The verifier either returns that the policy holds (green) or shows a counterexample to disprove it (gray). In the end, every policy has either been verified or disproved. As in data plane analysis, while control plane verification tools scale to the policies that we consider, enumerating all possible policies and checking them one by one in the above manner is prohibitive (see §8.2).

3 Our Approach: *Config2Spec*

In this section, we first present our key insight of combining the two baseline approaches from §2 and explain the reasoning behind it. Then, we provide an overview of the system (details are provided in the following sections).

3.1 Key Insight

We address the problem of mining a network specification by combining the baseline approaches and leveraging their respective strengths: data plane analysis is efficient at pruning policies, while control plane verification is efficient at validating policies. The key idea of our combination is to reduce the space of policies by sampling forwarding states and pruning policies using data plane analysis, and then running control plane verification to verify a small set of remaining policies.

This combination works well because many policies which do not hold are *dense violations*. That is, they are violated for many of the concrete environments captured by the failure model. For example, in our running example and the failure model $L_{symbolic} = L$ with $k = 1$ (up to one failure), the policy `waypoint(3, 1, p2)` only holds for the concrete environment in which all links are up, but the one from router `3` to `4`. Thus, by sampling any other concrete environment (e.g., $L_{down} = \{$`2-5`$\}, L_{up} = L \setminus L_{down}$), and computing all policies that hold for it, we can prune `waypoint(3, 1, p2)`.

On the other hand, there are *sparse violations*, which are policies that do not hold for the failure model, but are violated only by very few concrete environments. For example, in our running example and the same failure model, the policy `isolation(5, p1)` is violated only by two concrete environments: *(i)* $L_{down} = \{$`2-5`$\}, L_{up} = L \setminus L_{down}$ and *(ii)* $L_{down} = \{$`1-2`$\}, L_{up} = L \setminus L_{down}$. Unless we check these particular environments, this policy cannot be pruned by data plane analysis. Thus, we prune sparse violations during the step of control plane verification. Since the overall number of true policies and sparse violations is often significantly smaller than the number of concrete environments, control plane verification is an efficient solution for this.

3.2 The *Config2Spec* System

We build on this insight to design *Config2Spec* (Fig. 3), which takes as input the network configuration (of all devices) and a failure model and outputs the network specification.

Config2Spec runs in a loop which dynamically switches between the two approaches until the specification is mined. To achieve this, *Config2Spec* relies on three main components: *(i)* predictors, *(ii)* data plane analysis, and *(iii)* control plane verification. In addition, *Config2Spec* maintains two sets of policies, `cands` which overapproximates the specification, and `verified` which underapproximates it. We next explain these sets, the algorithm flow and the three components. We provide the full algorithm of *Config2Spec* in Appendix A.

Cands and verified *Config2Spec* keeps two sets: *(i)* `cands`, containing the current candidate policies, i.e., the policies that are known to hold or have not been pruned yet, and *(ii)* `verified`, containing the policies that are known to hold. `cands` initially contains all possible policies (blue area in Fig. 3), while `verified` is initially empty (green area in Fig. 3). We note that in practice, to avoid storing all policies in `cands`, only to prune many of them upon the first iteration of data plane analysis, *Config2Spec* directly initializes `cands` to the set of policies that holds for some concrete environment.

An invariant of the execution is that `cands` is a superset of the network specification, i.e., it contains at least all the policies that hold, while `verified` is a subset of it, i.e., it contains only policies that hold. *Config2Spec* terminates when these sets are equal – implying both equal the network specification – and then returns `verified`. Precision is ensured as *Config2Spec* does not miss any policy thanks to the invariant that `verified` contains only true policies (no false positives), while `cands` cannot miss a true policy (no false negatives).

Flow At each iteration, *Config2Spec* checks if `cands` equals `verified`. If so, it terminates. Otherwise, it checks two predictors to decide which approach is the more promising one to pursue: data plane analysis or control plane verification.

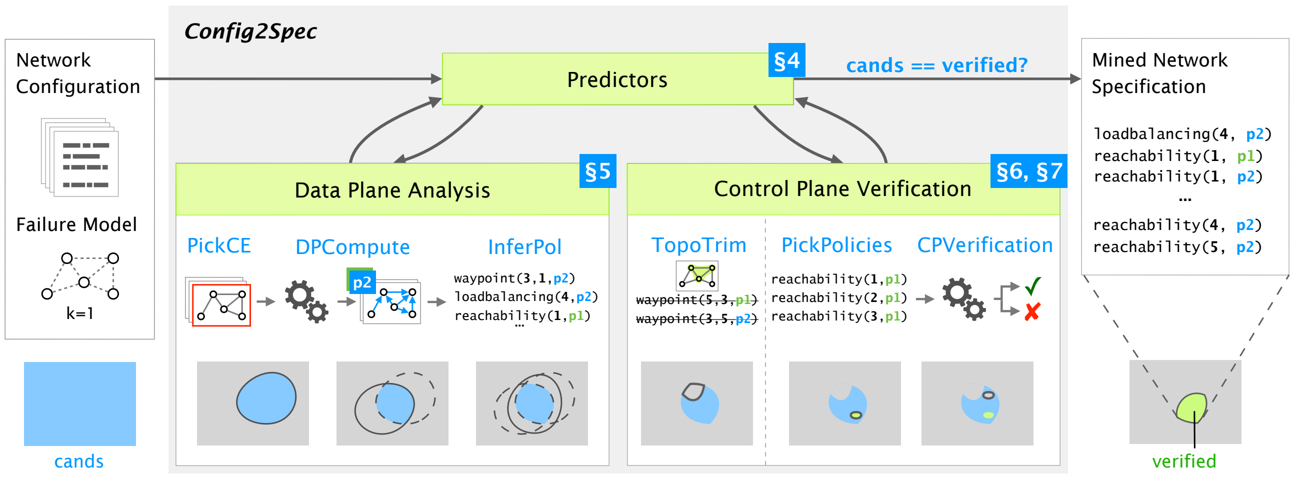


Figure 3: *Config2Spec* mines the specification from the network configuration and the failure model. It relies on three components: predictors, data plane analysis, and control plane verification. It maintains two sets: `cands`, consisting of the current candidate policies, and `verified`, consisting of the verified policies. During the execution, policies are removed from `cands` or added to `verified`. When `cands` equals `verified`, both equal the network specification, and then `verified` is returned.

Predictors (§4) We design two predictors to heuristically estimate which approach is likely to be more effective and dynamically transition between them. The predictors consider the execution times and the number of pruned and verified policies. The first predictor checks the effectiveness of each approach in classifying policies by measuring the time it needs to classify a single policy. The second predictor estimates the remaining time to mine the full specification.

Data plane analysis (§5) In every iteration of data plane analysis, *Config2Spec* samples a concrete environment, computes the policies that hold for it, and removes from `cands` any other policy. To sample a concrete environment, it executes `PickCE`, which employs a novel policy-aware sampler to find a concrete environment likely to prune more policies. Then, *Config2Spec* computes the data plane of that sample via `DPCompute`, which relies on prior tools (e.g., [13]). Next, it executes `InferPol` to compute all policies which hold for this data plane, and updates `cands` accordingly. Finally, *Config2Spec* checks whether all data planes have been analyzed. If so, it sets `verified` to `cands`, as the entire failure model has been covered and the full specification has been mined.

Control plane verification (§6) In each iteration of control plane verification, *Config2Spec* verifies a set of policies. For this, *Config2Spec* first executes `PickPolicies` to pick the next set of policies to verify. It then calls `CPVerification`, which relies on prior tools (e.g., [4]). The verifier either determines that all policies hold or returns a counterexample. In the former case, *Config2Spec* adds all the policies to `verified`, while in the latter case *Config2Spec* removes the ones violated by the counterexample from `cands`. Before the first iteration of control plane verification, *Config2Spec* invokes `TopoTrim` to reduce the verification overhead.

Topology-based trimming (§7) `TopoTrim` analyzes the topology and the failure model to trim (i.e., prune) policies which cannot hold regardless of the configuration (e.g., due to a lack of connectivity). It relies on graph algorithms to prune `reachability`, `waypoint`, and `loadbalancing` policies.

4 *Config2Spec*'s Predictors

In this section, we describe how *Config2Spec* dynamically decides whether to run the data plane analyzer or the control plane verifier. This decision relies on two predictors that capture the effectiveness of the approaches and the expected time remaining. Accordingly, *Config2Spec* infers which approach is more likely to make better progress. The predictors are: *(i)* the *Time-per-policy (TP) predictor*, favoring the approach more likely to classify more policies in a single execution, and *(ii)* the *Remaining-time (RT) predictor*, favoring the approach more likely to complete faster. If the predictors disagree on the approach, *Config2Spec* runs the data plane analyzer, we explain the reason for this choice shortly.

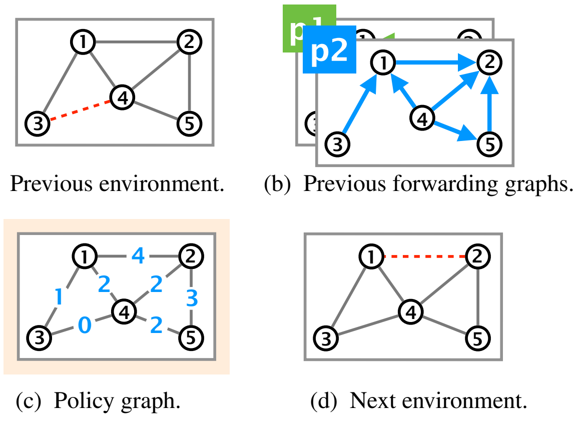
High-level behavior The predictors dynamically identify the different stages of the algorithm. In the beginning, sampling concrete environments is likely to provide the fastest progress, as at this stage the dense policies have not been pruned yet. Therefore, the TP predictor prefers data plane analysis initially. After most of the dense policies have been pruned, sampling environments may not significantly decrease the number of candidate policies anymore. At this point, the TP predictor starts to prefer control plane verification. Thus, the choice is then up to the RT predictor. It determines whether *Config2Spec* switches to control plane verification. If running data plane analysis for the remaining concrete environments is likely to be faster than running control plane verification on the remaining unclassified policies, the RT predictor prefers data plane analysis. Otherwise, it prefers control plane verification. This choice depends on the failure model: if it captures a small number of concrete environments, enumerating all of them can be faster than verifying the remaining set of candidate policies. In our running example and the failure model $L_{symbolic} = L$ and $k = 1$, this is the case. To conclude, the joint behavior of the predictors is to prefer control plane verification whenever *(i)* there is a large number of concrete environments and *(ii)* most remaining policies are true policies (i.e., part of the specification) or sparse violations.

Computation The predictors rely on statistics of the previous runs. The TP predictor is implemented by tracking two times: $T^{TP}_{analysis}$ and T^{TP}_{verify}, which record the average time to classify a single policy through analysis or verification (respectively). For $T^{TP}_{analysis}$, this time is computed by taking the ratio of the execution time of the last run of the data plane analysis and the number of policies which were pruned as a result of this analysis. For T^{TP}_{verify}, this time is computed similarly by taking the ratio of the execution time of the last run of the verifier and the number of policies which were classified by the verifier. The latter number is one of the following. If the verifier proved all policies hold, it equals the number of policies. Otherwise, if the verifier returned a counterexample, this number equals to the number of policies which were discovered as violations (i.e., the counterexample violated them). The TP predictor prefers the data plane analyzer if $T^{TP}_{analysis} < T^{TP}_{verify}$.

The RT predictor is implemented by tracking two (different) times: $T^{RT}_{analysis}$ and T^{RT}_{verify}, which record the execution time of a single run of the analyzer and verifier (respectively). The RT predictor prefers the data plane analyzer if the remaining time of the analyzer, obtained by multiplying $T^{RT}_{analysis}$ with the number of non-analyzed concrete environments is smaller than the remaining time of the verifier, given by multiplying $T^{RT}_{verifier}$ with the remaining number of unclassified policies.

Initialization To initialize T^{TP}_{verify} and T^{RT}_{verify}, *Config2Spec* executes the verifier on M policy sets (in our implementation, $M = 10$). It then sets T^{RT}_{verify} to the average execution time of the verifier, and T^{TP}_{verify} to the average ratio of execution time and policies verified or pruned. The estimates $T^{TP}_{analysis}, T^{RT}_{analysis}$ are initially 0, to guide *Config2Spec* to begin by data plane analysis. This captures our premise that initially data plane analysis is likely to classify more policies (the dense violations, which are the vast majority of the policies).

Windows To smoothen the behavior of the predictors, the times are averaged over the last N runs of the analyzer or verifier (in our implementation, $N = 10$).



(a) Previous environment. (b) Previous forwarding graphs. (c) Policy graph. (d) Next environment.

Figure 4: The policy graph is computed from the forwarding graphs of a previously analyzed concrete environment and guides us to an environment likely to prune more policies.

5 Data Plane Analysis

In this section, we present the key ingredients of running the data plane analysis in *Config2Spec*: the selection of the next concrete environment to analyze (`PickCE`), the computation of the data plane for that environment (`DPCompute`) and the inference of the policies from the data plane (`InferPol`).

5.1 Selection of Concrete Environments

At every iteration, one concrete environment is analyzed. The choice of this environment has a great impact on the overall runtime of the system. Thus, we design a sampling technique to pick the next concrete environment to prune a large number of policies from the set of candidates (`cands`). We call this technique *policy-aware sampling* as the next environment is picked based on the *policy graph*, a concept reflecting the current set of candidate policies, which we describe next.

Policy graph The *policy graph* for a given concrete environment is a copy of the network topology, augmenting the links with the number of policies that forward traffic along them. We say a `reachability(r,p)` policy forwards traffic along a link, if that link is part of a path in the forwarding graph of `p` from `r` to `p`. We define it similarly for the other policies. The policy graph allows us to identify the links on which large numbers of policies depend. Thus, we can pick a concrete environment in which these links are down. If the policies indeed hold only thanks to these links, they will be discovered as violations when analyzing this concrete environment.

We next define the policy graph. Given a network topology, a configuration, and a concrete environment, the policy graph extends the network topology with a mapping of links to weights (integers). The weight of a link represents the number of unclassified policies whose traffic is forwarded along that link. The weight is computed from the *forward-*

ing graphs of the concrete environment. Fig. 4 illustrates the concept of the policy graph using our running example (Fig. 1). Here, we are given an (already analyzed) concrete environment where all links are up, but the one between routers `3` and `4` (Fig. 4a). In this example, there are two destinations (`p1` and `p2`) and hence two forwarding graphs (Fig. 4b). For simplicity's sake, consider the following unclassified policies for destination `p2`: `reachability(i, p2)`, where `i` ranges over all five routers, and `loadbalancing(4, p2)`, which holds since router `4` has three paths to router `2` in the forwarding graph of `p2`. In this setting, the policy graph (Fig. 4c) maps, for example, link `1-3` to 1 (as only `reachability(3, p2)` depends on this link), link `2-5` to 3 (for `reachability(4, p2)`,`reachability(5, p2)` and `loadbalancing(4, p2)`), `1-2` to 4 (for `reachability(1, p2)`, `reachability(3, p2)`, `reachability(4, p2)` and `loadbalancing(4, p2)`), and `1-2` (which is down) to 0.

Policy-aware sampling Based on the idea of the policy graph, we design a policy-aware sampler for `PickCE`. The policy-aware sampler picks the next concrete environment to analyze based on the policy graph of the previously analyzed concrete environment and the current set of unclassified policies (`cands\verified`). This is done by selecting the links to add to L_{down} based on a probability distribution proportioned to the links' weights in the policy graph. The links' weights are computed by iterating over all unclassified policies (`cands\verified`) and counting, for each link, the number of policies that are forwarded along it. The probability distribution is needed to avoid getting stuck: a deterministic approach which adds the heaviest links to L_{down} can result in an oscillation between two concrete environments which already have been analyzed (we observed this phenomenon in practice). Adding non-determinism mitigates this issue, and in case it cannot, `PickCE` resorts to returning a random concrete environment which has not yet been analyzed. In the beginning, *Config2Spec* starts by analyzing the concrete environment in which all symbolic links are up.

For our running example and the policy graph in Fig. 4c, it assigns the link `1-3` to the probability $\frac{1}{14}$, `2-5` to $\frac{3}{14}$, and `1-2` to $\frac{4}{14}$. Assuming the usual failure model ($L_{symbolic} = L$ and $k = 1$), it then picks the next concrete environment by choosing one link that is down based on the distribution. For example, it picks the link `1-2` (Fig. 4d).

5.2 Analysis of a Concrete Environment

We now explain `DPCompute` and `InferPol`, which together compute all policies that hold for a given concrete environment and configuration.

The `DPCompute` algorithm executes two steps. First, for each router in the network, it computes the router's forwarding state. The forwarding state of a router is a list of destination prefix and next hop pairs. A pair `(p,w)` in the forwarding state of router `r` indicates that traffic reaching `r` for destination `p` is sent to router `w`. Computing the forwarding state of the routers is not trivial, however, there are solutions to efficiently compute them (e.g., [13]).

In the second step, `DPCompute` builds from the routers' forwarding states the forwarding graphs. It builds one forwarding graph for each equivalence class of destination prefixes (i.e., prefixes which can be captured via some prefix and have the same forwarding graph). The forwarding graph of a prefix `p` is a directed graph in which we have a link from router `r` to `w` if, according to `r`'s forwarding state, traffic for `p` is sent to `w`.

From the forwarding graphs, `InferPol` computes the policies by leveraging graph algorithms. For `reachability` and `waypoint` policies, it builds the *dominator tree* of all forwarding graphs. A dominator tree is a tree rooted at the destination of the forwarding graph. Its nodes are all routers that have at least one path to the destination. A router `a` is a child of a router `b` if *(i)* traffic from router `a` to the destination must pass through router `b` and *(ii)* for any other router `c` such that traffic from `a` must pass through it, traffic from `b` must also pass through it. `InferPol` infers a `reachability(r, p)` policy for every node `r` in the dominator tree of `p`. It further infers `waypoint(r,w,p)` for all routers `r` which are dominated by a waypoint `w` in the dominator tree of `p`. For `loadbalancing`, it computes the shortest paths in the network and infers `loadbalancing(r,p)` for routers `r` with multiple paths of the same cost available to reach destination `p`. For `isolation`, it infers `isolation(r,p)` for every router `r` and prefix `p` for which it has not inferred `reachability(r,p)`.

6 Control Plane Verification

Here, we present the two ingredients of the control plane verification in *Config2Spec*: the selection of policies to verify next (`PickPolicies`) and their verification (`CPVerification`).

CPVerification We begin with `CPVerification`, which takes as input a set of policies, the network configuration and the failure model. It checks whether all policies hold for any concrete environment meeting the failure model (for the given network configuration), or returns a counterexample.

Technically, the verifier symbolically encodes the configuration and the failure model as logical constraints: φ_{net} and φ_{fmodel}. The set of policies is encoded as a conjunction over formulas encoding the policies: $\varphi_{pols} = \bigwedge_{pl \in pols} \varphi_{pl}$. The verifier checks the satisfiability of $\varphi_{net} \wedge \varphi_{fmodel} \wedge \neg\varphi_{pols}$. If it is unsatisfiable, then all policies in *pols* hold. If the formula is satisfiable, then there is a counterexample, i.e., a concrete environment captured by the failure model, which under the given configuration violates φ_{pols} (i.e., at least one policy is violated). While the challenge of verifying network policies is not trivial, there are effective solutions (e.g., [4]).

PickPolicies This procedure takes the set of candidate policies (cands) and verified policies (verified) and returns the next set of policies to verify (from cands $\setminus$ verified). Since verifying is computationally expensive, the goal is to minimize the overall execution time of the verifier. By choosing a set of policies which have a dependency, the overall execution time of verifying them can be smaller than if they were verified one by one. Towards this goal, PickPolicies returns a maximal set of policies with the same destination prefix p.

We pick p arbitrarily, as once *Config2Spec* chooses to run the verifier, usually most policies are true policies.

Our grouping approach is always at least as good as verifying the policies one by one. The reason is that at each query to the verifier, at least one policy is classified. In the worst case, only one policy is classified as violation (if the verifier returned a counterexample which satisfies all policies but one). In a better case, several policies are classified as violation. In either of these cases, the violated policies are removed from cands, while the other policies in the set remain in cands (and will be verified in a later execution of CPVerfication). In the best case, all policies are classified as true policies. Namely, we can only gain from verifying multiple policies in the same execution of the verifier. Further, our grouping is maximal – grouping of policies with different prefixes is not helpful, as each prefix has a different forwarding graph, and so the verifier does not gain from grouping such policies.

7 Topology-based Trimming

In this section, we describe TopoTrim, a technique which reduces the load on the control plane verification by analyzing the failure model and the network topology. TopoTrim classifies policies as violations if their minimal connectivity requirements are not met under the given failure model.

TopoTrim is executed the first time *Config2Spec* chooses to run the verifier. It relies on the insight that some policies can be classified as violations directly from the network topology and failure model. For example, consider the network in Fig. 1 and the failure model with $L_{symbolic} = L$ and $k = 2$ (i.e., up to two link failures). We can infer that reachability(3, p1) cannot hold as 3 can become disconnected from the rest of the network if both links connected to it fail. For the same reason, any waypoint or loadbalancing policy where 3 is involved can be classified as violation.

To prune such policies, TopoTrim computes the $(k+1)$-edge-connected components of the topology for a failure model with k permitted failures. A $(k+1)$-edge-connected component is a set of nodes which remain connected even after removing any k edges. For example, for the network in Fig. 1 and the same failure model (where $k = 2$), the following routers are in a 3-edge-connected component: $\{1,2,4\}$.

There are efficient algorithms to compute $(k+1)$-edge-connected components, however they do not support links that must be up or down (L_{up} or L_{down}). To take these into account, TopoTrim first removes from the topology all links in L_{down}, updates k to $k - |L_{down}|$, and then, for each link in L_{up}, it adds k additional links between the routers to simulate that these routers are $(k+1)$-edge-connected. For example, for $L_{up} = \{(1,3)\}$, $L_{down} = \emptyset$ and $k = 2$, it adds two more edges between 1 and 3, so they are considered 3-edge-connected.

Based on this, TopoTrim classifies the following policies as violations (which are thus removed from cands). The policies reachability(r,p) and loadbalancing(r,p), for any router r and prefix p such that (r, r_p) is not in a $(k+1)$-edge-component, where r_p is the router attached to p. The policy waypoint(r,w,p) is classified as violation for any routers r and w and a prefix p such that *(i)* (r, w) is not in a $(k+1)$-edge-component or *(ii)* (w, r_p) is not in a $(k+1)$-edge-component, where r_p is the router attached to p.

8 Experimental Evaluation

In this section, we evaluate *Config2Spec* on multiple topologies to address the following research questions:

RQ1 How does *Config2Spec* scale to realistic topologies? We show that even for large networks with 158 routers and 189 links, it completes within 2.7 hours for OSPF configurations and 13.7 hours for BGP configurations.

RQ2 How does *Config2Spec* compare to the baselines? We show it improves the best one by up to a factor of 8.3.

RQ3 How do the domain-specific techniques contribute to *Config2Spec*? We show that *(i)* the policy-aware sampler leads to smaller candidate sets by up to a factor of 2 compared to random, and obtains them with fewer samples, and *(ii)* topology-based trimming and policy grouping reduce the queries by up to a factor of 2'500.

RQ4 Can *Config2Spec* be run on a real network configuration? We illustrate this on the Internet2 configuration.

Implementation *Config2Spec* is implemented in 5k lines of Python and Java code.[1] It computes the routers' forwarding states (§5.2) using Batfish [13], and verifies policies using Minesweeper [4]. We extended Minesweeper with the waypoint and loadbalancing policies. We note that while our implementation supports only configurations and features supported by these two third-party tools, our approach is not limited to specific configuration types or features.

Config2Spec takes as input the routers' configurations and a failure model. It outputs all policies that hold for the provided input. For large networks, we assume the network operator provides a list of devices that act as waypoints (e.g., middleboxes). In our experiments, we simulate it by randomly picking 20% of the routers to serve as waypoints.

Experiment setup To study how *Config2Spec* scales as a function of the topology size, we picked three topologies (small, medium, and large) from the Topology Zoo collec-

[1]Code is available at https://github.com/nsg-ethz/config2spec.

Topology	k	Config	Overall	DPA	CPV
BICS	1	OSPF	38.8 s	100%	0%
		BGP	68.3 s	100%	0%
	2	OSPF	228.8 s	30%	70%
		BGP	1'341.2 s	85%	15%
	3	OSPF	117.4 s	27%	73%
		BGP	319.7 s	14%	86%
Columbus	1	OSPF	398.0 s	100%	0%
		BGP	457.2 s	100%	0%
	2	OSPF	1'328.1 s	18%	82%
		BGP	6'772.0 s	17%	83%
	3	OSPF	907.0 s	27%	73%
		BGP	2074.1 s	18%	82%
US Carrier	1	OSPF	6'386.2 s	100%	0%
		BGP	6'813.4 s	100%	0%
	2	OSPF	10'528.4 s	15%	85%
		BGP	49'151.0 s	6%	94%
	3	OSPF	2'542.5 s	59%	41%
		BGP	5'873.3 s	34%	66%

Table 2: Execution time of *Config2Spec* as a function of the network topology, number of failures and configuration type.

Topology	k	Candidates	Specification	Percent
BICS	1	2'526.9	1'008.1	40%
	2	2'504.4	304.0	12%
	3	2'482.1	57.6	2%
Columbus	1	13'290.2	4517.1	34%
	2	13'150.4	350.4	3%
	3	13'271.0	27.2	0.2%
US Carrier	1	93'416.2	17'908.3	18%
	2	85'021.0	702.8	0.8%
	3	98'837.6	6.8	0.01%

Table 3: The number of candidate policies and the number of policies in the specification *Config2Spec* returns. Percent shows the fraction of the policies of all candidate policies.

tion [19]: BICS with 33 routers connected by 48 links, Columbus with 70 routers and 85 links, and US Carrier with 158 routers and 189 links. We used NetComplete [10] to synthesize OSPF and BGP configurations using its path-ordering specifications for 2, 4, 8 and 16 prefixes. For each configuration type and topology, we generated 5 configuration sets.

For each set of router configurations, *Config2Spec* computes all policies which hold, for all four policy types in Table 1. We consider three failure models, where k is 1, 2, or 3, and we fix $L_{up} = L_{down} = \emptyset$ and $L_{symbolic} = L$ (i.e., any link can be up or down). The reported results are averaged over these runs and the two configuration types (i.e., OSPF and BGP). We ran all experiments in virtual machines with 32 GB of RAM and 12 virtual cores running at 2.3 GHz.

8.1 Scalability of *Config2Spec*

We begin by studying how *Config2Spec* scales to realistic topologies. To this end, we ran experiments on all three topologies and three failure models, and measured the time *Config2Spec* spent on the data plane analysis part – including `PickCE`, `DPCompute` (which invoked Batfish), and `InferPol` – and the control plane verification part – including `PickPolicies` and `CPVerification` (which invoked Minesweeper). The other parts completed in negligible times and were thus ignored (e.g., `TopoTrim` completed within five seconds for US Carrier and less than a second for BICS).

Table 2 shows the overall execution time (*Overall*) and how it is split between data plane analysis (*DPA*) and control plane verification (*CPV*) as a function of the topology, the number of failures (k), and the configuration type (*Config*). For example, for the US Carrier topology with $k = 3$ and OSPF configurations, *Config2Spec* completed within 43 minutes, where 59% of that time was spent on data plane analysis.

The results show that even for the US Carrier topology with its 158 routers and 189 links, *Config2Spec* mined the specification in a reasonable time (within 2.7 hours, for OSPF, and 13.7 hours, for BGP). The results also demonstrate that the runtime mainly depends on the network size, secondly on the failure model, and lastly on the configuration type. This is expected: the larger the network, the larger the set of candidate policies and the set of concrete environments (whose size also depends on the failure model). In contrast to the effect of the network size on the execution times, the permissiveness of the failure model shows a different trend: execution times increase from $k = 1$ and $k = 2$, but drop for $k = 3$. This is thanks to the topology-based trimming (§7), which becomes very significant for $k = 3$ (or higher values of k). For the evaluated topologies, most router pairs are not 4-edge-connected, thus many policies are pruned. We provide more details on trimming in §8.3. The results show also that for $k = 1$, *Config2Spec* only performs data plane analysis. This is because the number of concrete environments is significantly smaller than the number of candidate policies throughout execution, leading the RT predictor to favor data plane analysis. Lastly, results show that for BGP configurations, the execution time is higher than for OSPF configurations. This is mainly due to Minesweeper, for which we observe a five to ten times increase in the verification time for BGP compared to OSPF.

Table 3 reports the number of candidate policies and the number of policies in the specification, for each topology and failure model, averaged across the different configuration sets and the configuration types. The reported number of candidate policies is the number of policies that hold for the first concrete environment picked by *Config2Spec* (*Config2Spec* always begins with data plane analysis). We consider this set as the initial set of candidates, rather than all instantiations

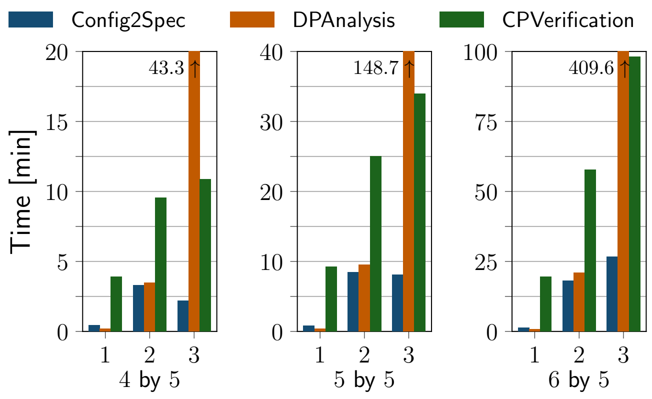


Figure 5: *Config2Spec* compared to the baselines of data plane analysis and control plane verification on grid topologies and different failure models. The bars of DPAnalysis and $k = 3$ are cut, and their maximum value is denoted next to them.

of the four policy types (Table 1), as the latter contains many policies which no concrete environment satisfies.

The results indicate that as the network size increases, the number of candidate policies increases, while the specification size (i.e., the number of policies that hold for all concrete environments) significantly drops. This demonstrates the challenge of *Config2Spec* to search in the large space of candidate policies for the small set of policies that hold.

8.2 Comparison to Baselines

We compare *Config2Spec* to the two baselines in §2: *(i)* a data plane analysis approach, which enumerates all data planes to infer the specification, and *(ii)* a control plane verification approach, which verifies the candidate policies one by one. As neither of the baselines scales to the larger networks considered in the last section, in this experiment, we use three grid topologies of sizes: 4 by 5, 5 by 5 and 6 by 5. We generated five sets of OSPF configurations per topology and used the failure model $L_{symbolic} = L$ with k ranging from 1 to 3.

Fig. 5 shows the execution time of each approach as a function of the topology and failure model. For $k = 2$ and $k = 3$, *Config2Spec* outperforms the baselines: the data plane analysis by 10.2x on average and up to 41.0x, and the control plane verification by 3.8x on average and up to 8.3x. For $k = 1$, data plane analysis is faster than *Config2Spec* because of *Config2Spec*'s setup time (i.e., the verification of few policies when initializing the predictors' times, see §4). Still, the overhead of *Config2Spec* is small (data plane analysis was faster on average by 24 seconds and by up to 37 seconds).

The results also show that both baselines have benefits. For less permissive failure models, data plane analysis performs better than control plane verification, whereas for permissive failure models it is the other way around. This demonstrates the advantage of the dynamic combination of *Config2Spec*.

8.3 Domain-specific Techniques

We next study how the domain-specific techniques improve *Config2Spec*'s performance. We study the following aspects: *(i)* how the policy-aware sampler (§5.1) helps reducing the number of concrete environments *Config2Spec* analyzes, and *(ii)* how topology-based trimming (§7) and policy grouping (§6) decrease the number of queries posed to the verifier.

Policy-aware sampler We compare the policy-aware sampler (called Policy-Aware) to a baseline which randomly picks a new concrete environment (called Random). We compare them by instantiating `PickCE` with each approach and running *Config2Spec* on the Topology Zoo topologies with the failure model $L_{symbolic} = L$ and $k = 3$, and with five sets of OSPF configurations and five sets of BGP configurations.

Table 4 shows the results. The first four columns show, for each approach, how many concrete environments were analyzed before *Config2Spec* transitioned to the verifier, and how many policies remained to verify (i.e., the percentage of remaining policies out of the policies that hold for the first sample). For example, for BICS, Policy-Aware required on average 36.4 samples before *Config2Spec* switched to verification, and at this point the size of the candidate policy set was reduced to 36.5% of the initial policy set (i.e., the set of policies which hold for the first sample).

Generally, the smaller the set of remaining policies (i.e., the closer the candidate set to the network specification is), the better. As a secondary goal, the number of analyzed concrete environments should be relatively small. Results indicate that Policy-Aware always obtains a better reduction in the size of the candidate set compared to Random. They also show that on average Policy-Aware typically required fewer samples than Random. However, we note that in 6 out of the 30 experiments, Random switched to verification before Policy-Aware did. This is not because Random made better progress. In contrary, the TP predictor decided to switch, as it observed that the concrete environments picked by Random were not effectively pruning policies anymore.

The next two columns of Table 4 provide more statistics. We checked, for each experiment, the relative size of the candidate sets for both approaches when *Config2Spec* with Policy-Aware transitioned to verification. For example, in one experiment using BICS, Policy-Aware transitioned to verification after 32 samples, and at that point the number of candidate policies was 970, while for Random, after 32 samples, there were 1'124 candidate policies, making the ratio 86.3%. In Table 4, Cands Ratio shows the average over the ten runs. We also checked how many additional samples Random required to reduce the candidate policies to (at most) the size obtained with Policy-Aware. For example, in that experiment for BICS, Policy-Aware required 32 samples to reduce the candidates to 970 policies, while Random required 43. Hence, Random needed 11 additional samples. In Table 4, Added Samples shows the average of this number. The re-

Topology	Policy-Aware Samples	Policy-Aware Candidates	Random Samples	Random Candidates	Cands Ratio	Added Samples	Policy-Aware PickCE	Policy-Aware DPAnalysis	Random PickCE	Random DPAnalysis
BICS	36.4	36.5%	42.1	39.5%	89.7%	45.7	22.1 ms	1.4 s	0.5 ms	1.3 s
Columbus	71.0	16.6%	79.0	26.4%	60.5%	109.0	63.1 ms	8.3 s	0.7 ms	7.7 s
US Carrier	113.8	9.6%	122.1	18.6%	51.6%	>500	358.2 ms	57.8 s	1.4 ms	51.6 s

Table 4: Comparison of the policy-aware sampler in `PickCE` (§5.1) and a random baseline. Samples is the number of samples before *Config2Spec* switched to verifier. Candidates is the percentage of remaining candidate policies at that point. Cands Ratio is the ratio of the candidate set sizes for Policy-Aware and Random when *Config2Spec* with Policy-Aware transitioned to verification. Added Samples is the number of samples Random needed to reduce the candidate set to the size of the candidate size with Policy-Aware. PickCE is the time to pick the next environment. DPAnalysis is the overall time to analyze a data plane.

sults indicate that Policy-Aware not only obtains a smaller candidate set, but reaches it significantly faster.

The last four columns of Table 4 show execution times: PickCE shows the execution time of the sampler, while DP-Analysis shows the overall execution time of a single data plane analysis (i.e., `DPCompute` and `InferPol`). Results show that while Policy-Aware takes more time than Random (as expected), the overhead is negligible compared to the overall execution time of the data plane analysis.

Topology-based trimming and policy grouping We next evaluate the topology-based trimming and policy grouping in reducing the number of queries to the verifier. We ran the experiments for the three topologies and the failure model with $k = 2$ and $k = 3$ (for $k = 1$, *Config2Spec* only performs data plane analysis §8.1). We measured how many queries to the verifier each technique saved. In every experiment, we recorded the number of policies *Config2Spec* had the first time it transitioned to the verification. This number, denoted B (for baseline), provides the number of queries to the verifier if we did not use either technique. We also recorded how many policies were pruned thanks to topology-based trimming. We count each policy that has been pruned as one saved query for the verifier, and denote the overall saved queries by T (for trimming). Also, we recorded how many queries were posed to the verifier (when employing policy grouping), and denote the number of queries by G (for grouping).

Fig. 6 shows the percentage of remaining queries after each optimization: $\frac{B-T}{B}\%$ for trimming and $\frac{G}{B}\%$ for policy grouping. For example, for BICS and $k = 2$, trimming pruned 51.1% of the policies. Policy grouping saved 41.5% and reduced the overall queries to the verifier to 9.6%. Overall, the reduction was 90.3%. The results show that the combination of trimming and policy grouping can reduce the number of queries to as little as 0.04%. Trimming is especially powerful for the larger topologies and for more permissive failure models ($k = 3$). The policy grouping also significantly reduces the number of queries to the verifier. The best case is for the largest network, where trimming reduced the number of queries to 1.15% and then policy grouping reduced it to 0.04%, compared to the baseline.

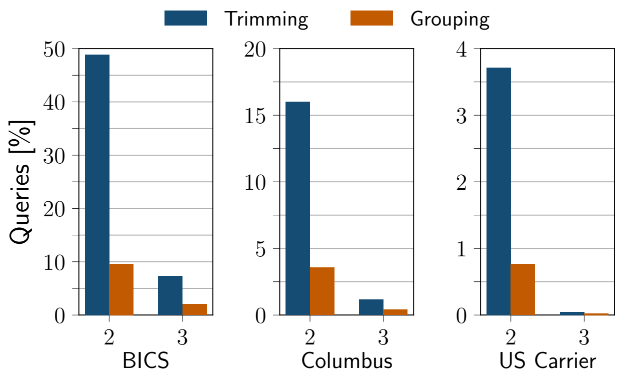


Figure 6: Reduction in the number of queries to the verifier thanks to topology-based trimming and policy grouping.

8.4 Running *Config2Spec* on Internet2

Finally, we demonstrate that *Config2Spec* can handle real configurations. For this, we took a publicly available configuration of the Internet2 network from May 2015 [8]. For Batfish to be able to parse this configuration, we had to remove multiple lines from it. Mostly, these parts concerned logging (e.g., `system dump-on-panic;`), anonymization leftovers (e.g., `Firewall Stanza Removed`) and other (for our purposes) irrelevant parts (e.g., `bfd-liveness-detection no-adaptation;`). For Minesweeper to be able to verify our queries, we had to remove parts of the BGP route-maps (community-matches and empty prefix-list matches). This does not affect the output, as we only mine the specification for internal prefixes, since no external peers are connected. In total, we had more than 90k lines of configuration. The topology consisted of 10 routers and 18 links. For a failure model with $L_{symbolic} = L$ and k from 1 to 3, *Config2Spec* required 32, 314, and 1'805 seconds to infer the network specification. It consisted of 3'962, 3'405, and 3'339 policies. The high number of policies, even for $k = 3$, stems from the fact that the five routers on the east-coast almost form a clique.

9 Related Work

In this section, we survey related work across five dimensions: specification mining, data plane analysis, control plane verification, network specification languages, and counterexample-guided inductive synthesis.

Specification mining Our work is inspired by works on specification mining [1], where high-level specifications are automatically inferred from low-level execution of programs. One example is Daikon [11], which dynamically detects program invariants (e.g., $x \neq 0$) by running the program and observing the values the program computes. For computer networks, Xie et al. [31] show how to compute the reachability specification for a given failure model based on the network's configuration. They compute the reachability upper bound – all policies that hold for at least one concrete environment – and lower bound – all policies that hold for all concrete environments. To scale, only an approximation of the bounds is computed. In contrast, *Config2Spec* computes the exact lower bound of reachability, as well as other policies, and thereby obtains a precise specification. Benson et al. [6] show how to mine reachability *policy units*, a high-level abstraction of pair-wise reachability, from network configurations for a single concrete environment. Like *Config2Spec*, it relies on data plane analysis. Unlike *Config2Spec*, failure models are not supported. Other works [7, 15] assess the complexity of managing the network and its overall health, i.e., the frequency of performance and availability problems, by analyzing its configurations.

Data plane analysis *Config2Spec* relies on a data plane analyzer. Several works exist and they differ mostly in their input. There are tools that require the forwarding state as input [16–18, 23], and others that compute the forwarding state from the network configuration [13]. These tools enable to check various properties, such as reachability and isolation, for the single forwarding state being analyzed.

Network verification *Config2Spec* also relies on a control plane verifier. Several works offer solutions for network verification, supporting different kinds of queries. Minesweeper [4] relies on an SMT-solver, and is currently the most general solution: it supports various properties (e.g., reachability, loop-freedom, router equivalence) and multiple (interacting) routing protocols. ERA [12] creates a unified control plane model that mainly allows to reason about reachability properties under multiple routing protocols. ARC [14] constructs an abstract graph representation of the data plane computation and supports various properties: reachability, isolation, waypointing and control plane equivalence. Many other tools focus on a single protocol such as Bagpipe [30].

Network specifications Many works introduce different network specification languages, varying in their expressiveness. Some allow to capture traffic classes at the path-level [3, 5, 27], while others use a higher-level abstraction describing traffic classes and high-level policies such as reachability and waypointing [24]. Despite the differences, *Config2Spec*'s output can be used by other tools, such as NetKAT [3], whose language can accommodate the policies we consider.

Counterexample-guided inductive synthesis (CEGIS) CEGIS is a technique in program synthesis in which examples guide the search for the target program [25, 26]. Technically, from an initial set of examples (which may be empty), the synthesizer proposes a candidate program consistent with the examples, and introduces it to a validator. The validator either confirms the candidate is the target program or returns a counterexample. The counterexample is added to the set of examples, guiding the synthesizer to look for a different candidate. *Config2Spec* can be seen as a synthesizer looking for (all) policies that hold for a given network configuration and failure model. Like CEGIS, it is guided by examples (the data planes) and a validator (the verifier). Unlike CEGIS, *Config2Spec* looks for *all* valid policies (and not a single one). This poses a greater challenge, both in terms of the search space and the burden on the validator. To cope, *Config2Spec* cleverly samples examples to prune the search space (without the help of the validator), trims and groups policies to save queries to the validator, and dynamically switches between sampling and verifying to expedite the search.

10 Conclusion

We introduced *Config2Spec*, a scalable approach for mining a network's specification from its configuration and a failure model. The key insight is to dynamically switch between data plane analysis and control plane verification. To scale further, we integrated three domain-specific techniques: *(i)* policy-aware sampling to pick concrete environments which are more promising for policy pruning, *(ii)* policy grouping to group queries and thereby reduce verification overhead, and *(iii)* topology-based trimming to prune policies infeasible for the given topology and failure model. We evaluated *Config2Spec* on different topologies and against two baselines. The results show that *Config2Spec* scales to large networks, unlike the baselines, and that our domain-specific techniques significantly contribute to the scalability.

Acknowledgements

We thank our shepherd Aditya Akella and the anonymous reviewers for the constructive feedback and comments. We are especially grateful to Ahmed El Hassany for his feedback and support with NetComplete.

References

[1] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining Specifications. In *ACM POPL*, Portland, OR, USA, 2002.

[2] Glenn Ammons, David Mandelin, Rastislav Bodík, and James R Larus. Debugging Temporal Specifications with Concept Analysis. In *ACM PLDI*, San Diego, CA, USA, 2003.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. In *ACM POPL*, San Diego, CA, USA, 2014.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *ACM SIGCOMM*, Los Angeles, CA, USA, 2017.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *ACM SIGCOMM*, Florianopolis, Brazil, 2016.

[6] Theophilus Benson, Aditya Akella, and David A. Maltz. Mining Policies from Enterprise Network Configuration. In *ACM IMC*, Chicago, IL, USA, 2009.

[7] Theophilus Benson, Aditya Akella, and David A Maltz. Unraveling the Complexity of Network Management. In *USENIX NSDI*, Boston, MA, USA, 2009.

[8] Min Cheng. Small. `https://github.com/jayvischeng/Small/tree/master/ServerData2`, 2015.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide Configuration Synthesis. In *CAV*, Heidelberg, Germany, 2017.

[10] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *USENIX NSDI*, Renton, WA, USA, 2018.

[11] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Elsevier Science of Computer Programming*, 69(1-3):35–45, 2007.

[12] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *USENIX OSDI*, Savannah, GA, USA, 2016.

[13] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. A General Approach to Network Configuration Analysis. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast Control Plane Analysis using an Abstract Representation. In *ACM SIGCOMM*, Florianopolis, Brazil, 2016.

[15] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. Management Plane Analytics. In *ACM IMC*, Tokyo, Japan, 2015.

[16] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking using Header Space Analysis. In *USENIX NSDI*, Lombard, IL, USA, 2013.

[17] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*, San Jose, CA, USA, 2012.

[18] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying Network-Wide Invariants in Real Time. In *USENIX NSDI*, Lombard, IL, USA, 2013.

[19] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The Internet Topology Zoo. *IEEE JSAC*, 29(9):1765–1775, 2011.

[20] Axel van Lamsweerde. Formal Specification: a Roadmap. In *ACM FOSE*, Limerick, Ireland, 2000.

[21] Claire Le Goues and Westley Weimer. Specification Mining With Few False Positives. In *TACAS*, York, United Kingdom, 2009.

[22] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking Beliefs in Dynamic Networks. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[23] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *ACM SIGCOMM*, Toronto, ON, Canada, 2011.

[24] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *ACM SIGCOMM*, London, United Kingdom, 2015.

[25] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *ACM PLDI*, Tucson, AZ, USA, 2008.

[26] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ACM ASPLOS*, San Jose, CA, USA, 2006.

[27] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A Language for Provisioning Network Resources. In *ACM CoNEXT*, Sydney, Australia, 2014.

[28] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks. In *ACM POPL*, Paris, France, 2017.

[29] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Synthesis of Fault-Tolerant Distributed Router Configurations. In *ACM SIGMETRICS*, Irvine, CA, USA, 2018.

[30] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *ACM OOPSLA*, Amsterdam, Netherlands, 2016.

[31] Geoffrey G Xie, Jibin Zhan, David A Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks. In *IEEE INFOCOM*, Miami, FL, USA, 2005.

Algorithm 1: *Config2Spec*(conf, $\mathcal{F}$)

```
Input  : conf: The network configuration.
         $\mathcal{F}$: the failure model (i.e., $L_{up}$, $L_{down}$, $L_{symbolic}$, $k$).
Output : A specification: the set of all policies that hold for the
         given configuration and failure model.

1  cands ← $allPolicies()$
2  verified, prevEnvs, lastFwds ← $\emptyset, \emptyset, \emptyset$
3  $T^{TP}_{verify}, T^{RT}_{verify}$ ← initVerificationTimes()
4  $T^{TP}_{analysis}, T^{RT}_{analysis} \leftarrow 0, 0$
5  totalEnvs ← $\sum_{j=0}^{k} \binom{|L_{symbolic}|}{j}$
6  while $cands \neq verified$ do
7      DP-RT ← $T^{RT}_{analysis} \cdot$ (totalEnvs − |prevEnvs|)
8      CP-RT ← $T^{RT}_{verify} \cdot$ |cands \ verified|
9      if $T^{TP}_{analysis} < T^{TP}_{verify}$ or DP-RT < CP-RT then
10         env ← PickCE($\mathcal{F}$, cands\verified, prevEnvs,
             lastFwds)
11         lastFwds, $T^{RT}_{analysis}$ ← DPCompute(env, conf)
12         pols = InferPol(lastFwds)
13         $T^{TP}_{analysis} \leftarrow (\text{cands} \setminus \text{pols} = \emptyset)\ ?\ \infty : \frac{T^{RT}_{analysis}}{|\text{cands} \setminus \text{pols}|}$
14         cands ← cands ∩ pols
15         prevEnvs ← prevEnvs ∪ {env}
16         if |prevEnvs| = totalEnvs then verified ← cands
17     else
18         pols ← PickPolicies(cands, verified)
19         cex, $T^{RT}_{verify}$ ← CPVerification(pols, conf, $\mathcal{F}$)
20         if cex = ⊥ then
21             verified ← verified ∪ pols
22             $T^{TP}_{verify} \leftarrow \frac{T^{RT}_{verify}}{|\text{pols}|}$
23         else
24             cands ← cands \ {p ∈ pols| cex violating p}
25             $T^{TP}_{verify} \leftarrow \frac{T^{RT}_{verify}}{\{p \in \text{pols} \mid \text{cex violating } p\}}$
26 return verified
```

A Main Algorithm of *Config2Spec*

Here, we present the main algorithm of *Config2Spec* (Algorithm 1). Our algorithm takes as input the configuration `conf` and a failure model $\mathcal{F}$ consisting of $L_{up}, L_{down}, L_{symbolic}$ and k. It outputs all policies which hold for this setting. The algorithm maintains the time estimates presented in §4 as well as a few sets. We next present these sets and the initialization of the time estimates, and afterwards the algorithm flow.

Main data structures The algorithm maintains four sets: `cands`, `verified`, `prevEnvs`, and `lastFwds`.

The `cands` set contains all policies that are still candidates for the network specification (i.e., unclassified policies and verified policies). That is, it is a superset of the network specification. When the algorithm terminates, `cands` is exactly the set of policies which hold. During the execution, policies which are discovered as violations are removed from `cands`. Initially, this set consists of all reachability, isolation, load balancing, and waypoint policies. Although we focus on these policies, our algorithm easily extends to any policy supported by the data plane analyzer and control plane verifier.

The `verified` set is the set of all policies that the verifier proved to be part of the network specification. That is, it is a subset of the set of policies which hold. When the algorithm terminates, `verified` is exactly the network specification. During the execution, policies which are discovered as true policies are added to it.

The `prevEnvs` set contains all previously analyzed concrete environments, while the `lastFwds` set contains the forwarding graphs of the last analyzed concrete environment, which is used to pick the next concrete environment.

Initialization of predictors' times As discussed in §4, our predictors rely on four time estimates: $T^{TP}_{analysis}$, $T^{TP}_{verifier}$, $T^{RT}_{analysis}$, and T^{RT}_{verify}. These are initialized as discussed in §4, where to initialize T^{TP}_{verify} and T^{RT}_{verify}, *Config2Spec* executes the verifier on M policy sets by running M times Line 18–Line 25, which are shortly explained.

Flow After initialization, *Config2Spec* runs in a loop which terminates when `verified` equals `cands`, indicating that both are equal to the network specification. At each iteration of the loop, *Config2Spec* first computes the predictors to pick between the data plane analyzer and the control plane verifier. The TP predictor checks $T^{TP}_{analysis} < T^{TP}_{verify}$. The RT predictor checks *DP-RT* < *CP-RT*, where *DP-RT* and *CP-RT* are the remaining times of the analyzer and verifier.

If the data plane analysis is chosen to be executed (Line 10–Line 16), *Config2Spec* invokes `PickCE` to pick the next concrete environment. It then calls `DPCompute`, to compute the forwarding graphs, and `InferPol`, to compute all policies from `cands` that hold for this environment. Afterwards, it updates the time estimates $T^{RT}_{analysis}$ (to the execution time of `DPCompute`) and $T^{TP}_{analysis}$ (to the execution time per policy which was pruned in this iteration). Then, it retains in `cands` only the policies that hold for the given environment and updates `prevEnvs` with the new environment. Finally, it checks whether there are still more concrete environments to analyze. If not, then `cands` contains only true policies, and so it sets `verified` to `cands`.

If the control plane verification is chosen (Line 18–Line 25), *Config2Spec* picks a set of policies to verify via `PickPolicies`. It then calls the verifier via `CPVerification`. The result is a counterexample `cex`, which may be ⊥, to indicate that all policies hold, or a concrete environment if some of the policies are violated. If `cex` is ⊥, all policies are added to `verified` and T^{TP}_{verify} is set to the ratio of the execution time and all policies (since all have been classified). If `cex` is not $\perp$, then *Config2Spec* removes from `cands` the policies which are violated by `cex`, and sets T^{TP}_{verify} to the ratio of the execution time and violated policies (since only they are classified).

Correctness We next discuss the correctness of *Config2Spec*. First, *Config2Spec* is precise. That is, it returns *all* policies which hold and *only* the policies which hold. The correctness argument relies on the data plane analysis and control plane verification being precise with respect to their tasks: the data plane analysis returns all and only those policies which hold for the given concrete environment, while the control plane verification returns a counterexample if and only if some of the given policies do not hold. With this assumption, we can prove the invariant that *(i)* `cands` always contains the network specification (i.e., the specification is a subset of it) and *(ii)* `verified` is always contained in the network specification. Because the algorithm terminates when these sets are equal, we get the guarantee.

Second, *Config2Spec* always terminates. For this, we rely on the data plane analysis and control plane verification to always terminate. We then make the claim that at each iteration either a new concrete environment is analyzed (guaranteed by `PickCE`) or at least one policy is classified (guaranteed by the control plane verification). Since the number of concrete environments and policies is finite, at some point either all policies are classified – at which point `cands=verified` and the algorithm terminates – or all concrete environments have been analyzed – at which point, *Config2Spec* sets `verified` to `cands` (Line 16), thereby terminating the algorithm.

Network Error Logging: Client-side measurement of end-to-end web service reliability

Sam Burnett[1], Lily Chen[1], Douglas A. Creager[3], Misha Efimov[1], Ilya Grigorik[1], Ben Jones[1], Harsha V. Madhyastha[1,2], Pavlos Papageorge[1], Brian Rogan[1], Charles Stahl[1], and Julia Tuttle[1]

[1]Google [2]University of Michigan [3]GitHub

nel-paper@google.com

Abstract

We present NEL (Network Error Logging), a planet-scale, client-side, network reliability measurement system. NEL is implemented in Google Chrome and has been proposed as a new W3C standard, letting any web site operator collect reports of clients' successful and failed requests to their sites. These reports are similar to web server logs, but include information about failed requests that never reach serving infrastructure. Reports are uploaded via redundant failover paths, reducing the likelihood of shared-fate failures of report uploads. We have designed NEL such that service providers can glean no additional information about users or their behavior compared to what services already have visibility into during normal operation. Since 2014, NEL has been invaluable in monitoring all of Google's domains, allowing us to detect and investigate instances of DNS hijacking, BGP route leaks, protocol deployment bugs, and other problems where packets might never reach our servers. This paper presents the design of NEL, case studies of real outages, and deployment lessons for other operators who choose to use NEL to monitor their traffic.

1 Introduction

Maintaining high availability is a matter of utmost importance for the operator of any popular web service. When users cannot access a web service, this may not only result in loss of revenue for the service provider but may also impact the service's reputation, causing users to shift to competing services. Therefore, it is critical that a web service operator detect and react in a timely manner when its service is inaccessible for any sizable population of users.

The primary challenge in doing so is that network traffic faces many threats as it traverses the Internet from clients to servers, most of which are outside the control of the service operator. Rogue DNS resolvers can serve hijacked results [2], ISP middleboxes can surgically alter traffic [27], bad router policy can silently drop packets [24], misconfigured BGP can take entire organizations offline [1], and more. Even though each of these issues is caused by systems that are not under the web service operator's control, the operator must bear primary responsibility for detecting and responding to them.

To address this challenge, a range of approaches have been developed over the years. For instance, server-side request logs (*e.g.*, the Apache web server's access.log and error.log files [31]) give fine-grained information about the success or failure of each incoming request. After annotating these logs with additional information, like the ISP or geographic location of the end user, operators can identify when interesting populations of end users are all affected by the same reliability issues [5, 4]. Alternative approaches rely on a dedicated monitoring infrastructure comprising a globally distributed set of vantage points. These approaches either actively probe the service to detect unreachability [6, 20] or passively monitor BGP feeds to identify routing issues [7].

Unfortunately, these existing solutions suffer from two fundamental limitations.

- First, they are typically capable of only detecting large, systemic outages. For example, with server-side monitoring, a major problem (*e.g.*, a global outage of a major service, or a regional outage affecting a large enough region) might show up as a noticeable drop in total request volume [28], but operators typically only learn of smaller problems when users manually report them.[1] These user reports are often frustratingly vague, and collecting additional information from nonexpert users is next to impossible, so investigation may take hours or even days.
- Second, and more importantly, existing approaches are incapable of precisely quantifying how many clients are affected, if any. For example, active probing from dedicated probing infrastructure can only probe from a handful of locations relative to the number of real users, and probe traffic is not always representative of what actual users experience (*e.g.*, probers may not use real web browsers and might receive different network configuration than actual end users). Without knowing how many users are affected, the operator is unable to judge whether it should prioritize the troubleshooting of a detected problem over other ongoing issues that deserve its attention.

To overcome these challenges in detecting and scoping instances of service unreachability, we need a system that (1)

[1]While sites like https://downdetector.com crowdsource such reports, historically we have often only learned of problems via social media.

passively monitors *actual* end user traffic to any target web service; (2) has visibility into reliability issues regardless of where on the end-to-end path they occur; and (3) requires little to no custom engineering work on the part of the operator.

With these goals in mind, we have designed, implemented, and deployed NEL (Network Error Logging). The key intuition behind NEL's design is that clients have ground truth about their ability to access a web service. Therefore, NEL leverages end users' browsers to collect reliability information about the service. NEL is implemented directly in the browser's network stack, collecting metadata about the outcome of requests, without requiring any custom per-site JavaScript instrumentation. The browser then uploads these reports to a redundant set of collection servers, which aggregates reports from users around the globe to detect and scope network outages.

This paper describes our contributions, based on the following three tracks:

First, we present our solutions to the various engineering and policy challenges that arise in using client-side data to detect reachability issues. When clients are unable to talk to a service, how do we still ensure successful collection of unreachability reports from these clients? What should uploaded reports contain so that they aid in diagnosing and estimating the impact of outages? How do we prevent abuse of the system, given that clients are free to upload fraudulent reports and service operators can attempt to learn about clients' reachability to other services? Our primary consideration in answering these questions has been to preserve user privacy; we ensure that NEL does not reveal more about clients than what service operators would learn during normal operation.

Second, we describe our experiences in using NEL to monitor reachability to Google's services since 2014. In that time, as we relay in Section 4, it has been instrumental in detecting and mitigating a wide variety of network outages including routing loops, BGP leaks, DNS hijacks, and protocol misconfigurations. In particular, without NEL, it would have been hard, if not impossible, to estimate the number of clients affected by each outage. Thus, NEL has proved invaluable in helping us identify which problems warrant immediate investigation due to their large impact and which ones we can afford to ignore or attend to later.

Third, after several years of experience with an initial implementation that could only monitor Google services, we describe our recent efforts to promote this capability as a new proposed W3C standard [11]. Standardizing this work has two benefits: (1) it allows all service operators to take advantage of this new collection ability, and (2) it allows operators to collect reliability data from any user agent that complies with the standard, and not just Chrome.

NEL is not a panacea for painstaking problem detection and diagnosis. It cannot report problems when clients are completely disconnected from the network or cannot reach at least one of a redundant set of collectors. It reports only coarse-grained summaries about entire requests, and is no substitute for lower-level network diagnostics like traceroutes and packet captures. (For example, it can detect when clients experience connection timeouts, but cannot tell you much more about why.) Nonetheless, NEL has proven a valuable tool for detecting and scoping network outages that are invisible to other monitoring infrastructure.

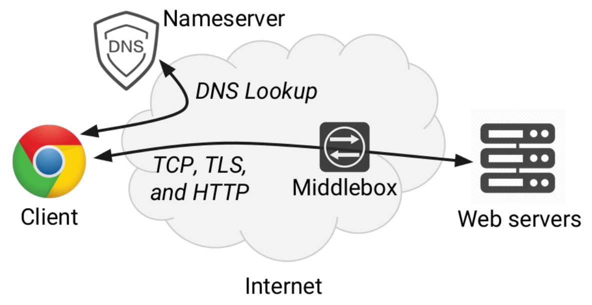


Figure 1: Steps and entities involved in enabling a client to access a web service.

2 Background and Motivation

We begin by listing several causes that may render a web service inaccessible, solutions that exist to detect service reachability problems, and the limitations of these solutions that motivated us to develop NEL.

2.1 Causes of service inaccessibility

As Figure 1 shows, a typical communication between a client and a web service offered over HTTPS involves the following steps: the client performs a DNS lookup of the service's hostname, establishes a TCP connection to the IP address it obtains, performs a TLS handshake with the server, and then sends its HTTP request.[2]

Given these steps, a client may be unable to communicate with a web service due to any of the following reasons:

- **DNS failure:** The client will be unable to execute any of the subsequent steps if its attempt to resolve the service's hostname fails. This can happen either if the nameserver that the client uses is unresponsive, or if the service provider's DNS setup is misconfigured.
- **DNS hijack:** The client could get an incorrect IP address in response to its DNS request if either the nameserver that the client uses is compromised or if the client is compromised to use a malicious nameserver.
- **IP unreachability:** When the client does get a correct IP address, the Internet may be unable to route packets from the client to that IP, either due to problems with BGP (*e.g.*, misconfiguration or convergence delays) or because of active blocking by network operators.

[2]Note that this only considers the user's communication with a "front-end" server. Modern services typically require many back-end service calls to generate the final response, which are hidden behind the interaction with the front-end server, and invisible to the client (and by extension, NEL).

- **Prefix hijack:** Alternatively, if the prefix which contains the IP address has been hijacked, the client's requests will be directed to servers not controlled by the service provider. The client will hence be (hopefully) unable to complete TLS connection setup.
- **Faulty middlebox:** When IP-level reachability between the client and the service is functional, the client's attempt to connect to the service may still fail due to a misconfigured or malicious middlebox enroute.
- **Service faulty/down:** Lastly, even if the client's DNS lookup succeeds and both IP-level and TLS-level connectivity are unhindered, the client will be unable to access a service that is down or misconfigured.

2.2 Existing solutions and limitations

To detect unreachability caused by the above-mentioned problems, a wide range of solutions have been developed over the years. This body of prior work falls broadly into four categories.

Monitoring from distributed vantage points. A popular approach is to monitor reachability from a dedicated set of distributed vantage points. Solutions that use this approach either actively probe services from devices that mimic real clients [3, 6]—probing can either be at the application level (*e.g.*, in the form of HTTP requests) or at the routing level (*e.g.*, in the form of traceroutes)—or passively monitor BGP updates (*e.g.*, to detect prefix hijacks [22, 19]), or use some combination of the two [20, 35]. Such approaches can detect problems that have wide impact. Localized problems are, however, likely to go unnoticed because a set of vantage points typically cannot match the global coverage of a service's clients. Moreover, when unreachability is detected, it is hard to estimate how many real clients are affected.

Monitoring service logs. To ensure broad coverage, service providers can monitor their service's usage either by analyzing server-side logs or by augmenting pages on their site with JavaScript that performs and uploads client-side measurements (popularly referred to as Real User Monitoring, or RUM for short [9]). With either strategy, operators must infer reachability problems from the *absence* of traffic. Requests from affected users will never arrive at the server and therefore are absent from server-side logs, whereas users unable to fetch even the HTML of a web page will not execute any JavaScript included on the page, even if cached. For any population of users, a significant drop in requests compared to what is typically expected (given historical traffic volumes) may indicate a reachability problem being experienced by these users. The challenge here is: how to distinguish between localized reachability problems and intrinsic volatility in traffic volumes? While traffic in aggregate across a large population of users is typically fairly predictable (*e.g.*, same from a particular hour in a week to that hour next week), the smaller the subset of users considered, the larger the unpredictability. As a result, drops in traffic volumes for small populations of users are not adequate evidence for service operators to take action.

Data source	**Enable timely detection**	**Detect localized outages**	**Estimate # of affected clients**
Distributed monitoring infrastructure	✓	×	×
Service logs	✓	×	×
Backscatter traffic	×	×	×
User reports	×	✓	×
NEL	✓	✓	✓

Table 1: Properties satisfied by different approaches for detecting service reachability problems at scale.

Monitoring backscatter traffic. Similar limitations exist with solutions that rely on backscatter traffic—traffic that clients send to unused portions of the IP address space—to detect reachability issues [12]. Here too, one has to infer (for example) censorship based on the absence of traffic. Consequently, problems can be reliably detected only when they are large in scope. Moreover, even when backscatter traffic shows an absence of traffic from a large population, those users likely cannot reach *any* IP address.

Leveraging user reports. An approach which can detect localized problems, unlike the previous categories of solutions, is to rely on complaints/reports from users. However, such solutions are typically incapable of detecting reachability problems in a timely, reproduceable, representative, and consistent manner. For example, users in some regions may be less likely to notify service operators about problems, due to language or cultural barriers.

Table 1 summarizes the limitations of these existing solutions. The overriding one, which motivated our development of NEL, is the inability to accurately estimate *how many clients are currently affected by an outage*. In our experience, a high confidence estimate of the scope of a problem is the top criterion to warrant investigation by a service operator. Lacking such data, it is hard to triage and prioritize human effort to troubleshoot the large number of issues that a service provider has to deal with at any point in time.

3 Design

This section presents the design of NEL, our browser-based mechanism for collecting information about a web service's availability, as seen by its clients. By using the client as the vantage point, the operator gains *explicit* information about the impact of reliability problems, rather then having to *estimate* the impact based on either the absence of traffic or by extrapolating from a small set of clients.

Google has twice implemented the ideas behind NEL: first as a proof of concept that could only monitor reachability to Google properties, and again as a proposed public W3C stan-

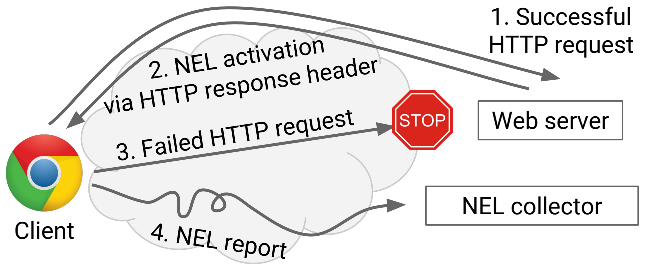

1. Successful HTTP request
2. NEL activation via HTTP response header
3. Failed HTTP request
STOP
Web server
NEL collector
Client
4. NEL report

Figure 2: When a client successfully communicates with a service, collection of client-side reports is activated via a NEL policy in the service's response headers. The client reports the success and failure of its subsequent requests to that service to collectors referenced in the NEL policy.

dard [11] available to all service operators. Here, we focus on the latter due to its general availability. We also summarize important differences between the two implementations.

3.1 Approach and challenges

When stated at a high level, NEL's approach is fairly simple and intuitive: clients upload reports summarizing their ability to either successfully or unsuccessfully access target services. In aggregate, such reports enable the provider to piece together the ground truth as to how many, and which, of its clients are unable to access its service.

Realizing this approach in practice requires us to answer several questions:

- How do clients know which services to collect reliability information about?
- What should clients include in reports they upload?
- In order to reliably report that information, where should clients upload reports to?

Before describing our answers to these questions (summarized in an illustration of NEL's architecture in Figure 2), we first list the properties required of such a system, which motivate our design choices.

3.2 Security, privacy, and ethics

When designing NEL, we have to balance collecting enough information to enable quick detection of reliability issues versus satisfying the security and privacy expectations of both service owners and their end users. There are four principles in particular that we must follow:

1. We cannot collect any information about end users, their device/user agent, or their network configuration, that the server does not already have visibility into. That is, we should not collect *new information* relative to existing server logs; only existing information in a *different place*.
2. We can only collect information about requests that user agents issue when users voluntarily access services on the Web. We cannot issue requests in the background (*i.e.*, outside of normal user activity), even though this prevents us from proactively ascertaining service reachability.
3. End users can opt out of collection at any time, either globally or on a per-site basis. Support for respecting opt-outs must be implemented by NEL-compliant user agents, so that users do not need to trust service providers for opt-outs to take effect.
4. Modulo that end-user opt-out, it is *only* the site owner who gets to decide whether reports are collected about a particular site, and if so, where they are sent. Third parties (including browser vendors) must not be able to use NEL to monitor sites that they do not control.

These principles have clear ramifications on the design of the system, as we discuss below in our description of NEL's design; Table 2 provides a summary.

3.3 When do clients generate reports?

Configuration via response headers. How does a user agent know which requests to collect reports about? An operator needs a way to instruct client browsers to collect reports about requests to services they control, along with any configuration parameters about that collection.

HTTP response headers provide exactly what we need: service operators insert policy configuration headers (Report-To and NEL) into their outgoing responses; Figure 3(a) shows an example. The user agent's network stack intercepts these policy headers as part of the normal processing of the response. NEL is limited to secure connections—HTTPS connections with validated certificates—ensuring that (in the absence of a subversion of the certificate trust validation mechanism) third parties cannot inject NEL policies into the responses of servers not under their control.

If an attacker does somehow subvert connection security, they could inject NEL's HTTP headers and can obtain NEL monitoring data. For example, a malicious or compromised CDN provider could siphon NEL logs off to their own collector. But such an attacker could obtain the same data by other means (*e.g.*, by injecting additional JavaScript).

Scope of activation. We need to ensure that client-side collection of NEL reports does not allow third parties to collect information about sites they do not control. The cleanest way to do this is to follow the existing Same-Origin Policy (SOP) [8], and to have report collection be scoped to the origin (domain name, scheme, and port) of the request. That is, collection would be activated (or deactivated) for all requests to an origin; any collection policy configured for origin A would have no effect on requests to origin B, even if both origins happen to be owned and operated by the same entity.

Note that NEL does not preclude user agents from generating NEL reports even when the user is in private browsing mode. In such cases, any service's server-side logs do reflect the user's use of its service. NEL is simply collecting the same information at the client and uploading it via redundant paths to the same service provider. As we describe later in this section, the contents of any NEL report ensure

Goal	Approach	Refinements/Experience
Securely **activate client-side collection** of reachability reports	• Include NEL policy header in HTTP responses • Enforce Same-Origin Policy • Limit to HTTPS connections	• With include_subdomains option, one successful communication with an origin suffices for a user agent to start collecting (some) reports for all subdomains of that origin
Ensure report content **preserves user privacy**, yet **aids diagnosis**	• Only include information that is visible to service during normal operation • Limited set of hierarchically organized error types	• Upload reports even for successful requests to establish baseline request rate and to reduce burstiness of collection workload
Enable secure and timely **collection of NEL reports** from clients	• Causes that can render collectors unreachable should be disjoint from those that can affect the service's reachability • Client downgrades report (removing sensitive information) if service's IP address is not one from which it has received NEL policy for this origin	• Specify weight and priority per collector to balance load and to enable failover across collectors • Configurable sampling rates to reduce load on both clients and collectors • Host collectors on cloud infrastructure to deter censorship of report collection

Table 2: Summary of the approach taken in NEL to address different design decisions, along with experience-based refinements.

that the service provider can glean no additional information about its users than it can from its server-side logs.

Preventing DNS rebinding attacks. On its own, SOP is not enough to prevent a malicious actor from using NEL to learn about the reachability of origins that they do not control. Consider a *rebinding* attack, where an attacker who owns bad.example wishes to learn about the availability of good.example. They start by configuring DNS to resolve bad.example to a server that they control (using a short TTL), and getting an end user to make a request to bad.example. The server returns a NEL policy instructing the user agent to send NEL reports to a collector run by the attacker. They then update DNS to resolve bad.example to good.example's IP address(es), and cause the user to make another request to bad.example. Even though the request *looks* like it will go to the original bad.example server, it will instead be routed to good.example's server; if there are any errors with the connection, NEL reports about those errors will be sent to the attacker's collector. In this way, the attacker has been able to collect error reports about good.example, even though they do not own it. This kind of attack could also be used to port scan an internal network, by repeatedly rebinding bad.example to several different internal IP addresses.

NEL prevents such attacks by having user agents *downgrade* the quality of a report when the server IP address that a user agent contacts is not one from which it previously received the NEL policy header for this origin. In this case, instead of reporting whether the request succeeded or not (and the error type if not), the report simply states that the user agent's DNS lookup yielded a different IP address. This information is safe to report to the attacker, since it is information that they already knew; and, because it relates to what addresses bad.example resolves to, the attacker is actually the legitimate party to deliver this information to. Note that this can limit the utility of NEL's reports for domains that resolve to many IP addresses (*e.g.*, CDNs).

Subdomain reports. A consequence of activating NEL via headers in HTTP responses and enforcing SOP is that NEL cannot help an operator discover a client's inability to access their service unless the client has successfully communicated with the service at least once in the past. To minimize the impact of this constraint on service providers, a NEL policy can include the include_subdomains field, which tells the user agent to collect and upload reports for both the origin as well as all of its subdomains.

At first glance, this is a clear violation of SOP: there is no guarantee that the web sites hosted at each subdomain are owned by the same entity.[3] To maintain our privacy properties, a user agent can only use an include_subdomains policy to report *DNS errors* about requests to a subdomain, since the author of the policy has only been able to establish ownership of that portion of the domain name tree. Subdomain reports about successful requests, and about any errors that occur during or after connection establishment, are *downgraded* to reports only containing information visible in DNS. Such reports suffice for the service provider to discover unreachability due to DNS misconfiguration, *e.g.*, the provider may have forgotten to add a DNS entry for a new subdomain.

3.4 What do clients upload?

Report content. The most important part of a NEL report (Figure 3(b) shows an example) is the type, which indicates whether the underlying request succeeded or failed. If the request succeeded, the report's type will be `ok`; if it failed, it will describe what error condition caused the request to fail. The full set of predefined error types is given in the specification [11, §6]; examples include `http.error`, `dns.name_not_resolved`, `tcp.reset`, and `tcp.timed_out`. These predefined types are categorized hierarchically, so that one can find, for example, all TCP-related failures by looking for any type that starts with `tcp`.

We have found it useful to collect NEL reports even for successful requests, despite the fact that these requests also show up in server-side logs. Reporting on successful re-

[3]Consider a PaaS cloud offering like Google App Engine, where independent cloud applications are hosted at subdomains under appspot.com.

```
Report-To: {
  "endpoints":  // Try to send reports to one of these URLs
    [{"url": "https://collector1.com/upload-nel"},
     {"url": "https://collector2.com/upload-nel"}],
  "group": "nel",  // The name of this group of endpoints
  "max_age": 300  // This set of collectors expires in 5 minutes
}
NEL: {
  "failure_fraction": 1,  // Report all failed requests
  "success_fraction": 0.25,  // Report 25% of successful requests
  "include_subdomains": false,  // Don't report for subdomains
  "report_to": "nel",  // Report to a collector in this group (above)
  "max_age": 300  // This NEL policy expires in 5 minutes
}
```

(a)

```
[{
  "age": 60000,  // The report was 1 minute old when uploaded
  "type": "network-error",  // This is a NEL report
  "url": "https://example.com/about/",  // The URL the client requested
  "user_agent": "Mozilla/5.0",  // The client's User-Agent header
  "body": {
    "type": "tcp.timed_out"  // The connection timed out
    "phase": "connection",  // The request failed during handshake
    "server_ip": "203.0.113.75",  // The client tried to connect to this IP
    "sampling_fraction": 1.0,  // This report had a 100% chance of collection
    "protocol": "h2",  // The request was made using HTTP/2
    "method": "GET",
    "referrer": "https://example.com/",  // The HTTP Referer
    "elapsed_time": 45000,  // Lifetime of the request in milliseconds
  }
}]
```

(b)

Figure 3: Examples of (a) a service's use of NEL headers in its HTTPS response to activate report collection by a user agent, and (b) a report uploaded by a user agent. Comments are not included in real headers and reports.

quests lets us directly compare error ratios from successful and failed reports, without having to join the NEL logs with server-side logs. Comparing successful reports to web server logs also lets us estimate the relationship between NEL report volumes and actual request volumes.

In addition to the type, NEL reports can only contain a fixed set of additional information, as defined by the public specification. This helps ensure that implementors do not accidentally include additional information that would violate our desired privacy properties. In authoring the specification, our primary constraint when determining which fields to include is to ensure that every field in the report contains information that the server can already see during its normal processing of the request.

Given this constraint, NEL reports include basic information about the request: URL (with user credentials and fragment identifiers removed), HTTP request method (`GET`, `POST`, etc.), application and transport protocol (HTTP/1.1, HTTP/2, QUIC, etc.), User-Agent string, and referrer. The reports also include the HTTP status code of the response, if one was received, and how long the request took to complete.

Reports also contain the IP address of the server that the user agent attempted to connect to. For most requests, this is the public IP address of the service's front-end server that directly accepts incoming connections from end users. Inclusion of this IP address in reports is crucial to enable detection of DNS hijacking; though the error type in the report may indicate successful TCP connection setup, the server IP address mentioned in the report will not be one used by any of a service's front-end servers.

As mentioned above, there are several situations where a NEL report is *downgraded* for privacy reasons; for instance, when the server IP of the request is not one that the corresponding NEL policy was received from. In these cases, any privacy-sensitive fields are modified or removed from the report, to maintain the property that the report only contains information that the policy author already had access to. The NEL specification [11] contains more detail about precisely which fields are modified or removed, and when.

What do reports *not* contain? There are many details about the client that we explicitly exclude from NEL reports, even at the expense of hampering diagnosis. For example, if a user agent is using a client-configured proxy server, the IP address that the user agent attempts to connect to would be the IP address of that proxy server. Since that proxy configuration is not intended to be visible to the server, we cannot include the IP address in the report. Note that this restriction only applies to proxies configured *by the end user*. If their ISP is using a transparent proxy for all of its customers' requests, any individual user agent won't easily be able to detect this. That means that the server IP reported by the user agent will still be the actual address of the origin server, while the client IP address seen by the server and any NEL collectors will be the address of the transparent proxy.

Similarly, we cannot include the IP address of the DNS resolver that the client uses. For DNS-related network outages, this would be useful information to collect, since it would help the service owner determine whether a rogue or misconfigured DNS resolver is at fault for an outage; however, since this information is not visible to the server when processing a request, we cannot include it in a NEL report.

NEL reports also do not include details about HTTP requests that are immaterial to diagnosing reachability problems. For example, user agents exclude cookies and URL parameters from reports. A NEL report does include the full path that a request was issued for, not just the hostname to which the request was issued. We have not found much use for this information so far, but it may prove useful to an operator whose service configuration varies across different pathnames under the same origin.

Sampling rates. For high-volume sites, it is undesirable to have clients generate NEL reports about *every* attempted request, since that could double the number of requests a client must make and would require the site's collection infrastructure to be able to deal with the same full volume of request traffic as the actual site. NEL allows service operators to define a *sampling rate*, instructing user agents to only generate reports about a random subset of requests. Moreover, they can provide separate sampling rates for successful and failed requests. Typically, one will want to configure a very high sampling rate for failed requests, since those requests are more operationally relevant and more important to collect as much information about as possible. The utility of collecting reports for successful requests is largely to estimate their total number, so lower sampling rates (*e.g.*, 1–10%) are typically sufficient. Each NEL report includes the sampling rate in effect when that particular report was generated, which allows collectors to weight each report by the inverse of its sampling rate when determining totals.

To reduce overhead, it may be tempting to adaptively vary the sampling rate over time. However, the need to increase sampling rates will arise precisely when an outage occurs. At that point, it will be infeasible for the service provider to update the NEL policy being used by affected clients.

Google uses a 5% sampling rate for successes and 100% for failures. We chose these numbers based on our experience working with NEL: (1) we find a lower sampling rate dilutes our data too much when examining small user populations (*e.g.*, when investigating outages in small ISPs), and (2) we want a relatively consistent report traffic volume, rather than massive spikes in load during major outages.

3.5 Where do clients upload reports to?

Once a user agent has generated reports about requests to an origin, those reports must somehow be sent back to the service operator's monitoring infrastructure. To do this, the service operator defines a set of *collectors*, giving an upload URL for each one (see Figure 3(a)). Since the set of collectors is defined in the NEL policy included in HTTP response headers, service operators have full control over where NEL reports about their origins are sent to.

User agents will periodically batch together reports about a particular origin, and upload those reports to one of the origin's configured collectors. The report upload is a simple `POST` request, with the JSON-serialized batched report content in the request payload.

Report uploads are subject to Cross-Origin Resource Sharing (CORS) [32] checks. If the origin of the collector is different than the origin of the requests that the NEL reports describe, the user agent will perform a CORS preflight request to verify that the collector is willing to receive NEL reports about the origin. If the CORS preflight request fails, the NEL report will be silently discarded. Reports are only uploaded over HTTPS to prevent leaking their content to passive in-network monitors.

Collector failure modes. For an operator to detect outages in a timely manner, it is crucial that clients be able to upload NEL reports even when they are unable to reach the monitored service. This requires that the collection path differ from the request path in as many ways as possible. As a consequence, not only must the collectors be hosted differently than the monitored service, but it is desirable that there be significant hosting diversity among those collectors. Examples of the ways in which collectors might differ from the monitored service include: different IP address (to learn about the service's IP being unreachable); different version of IP (if IPv4 is reachable, but not IPv6); different AS number (to account for BGP/routing issues); different transport protocol (*e.g.*, for QUIC-specific problems); and different hostname, registrar, and DNS server[4] (if the service's nameserver is unreachable). Later, in Section 4, we recount the kinds of collector diversity that have proved to be most valuable in our experience.

Given the effort necessary to ensure that collectors for a service do not share the same failure modes as the service itself, one may wonder whether the collectors could be used to improve the availability of the service, beyond collecting evidence of its (un)reachability. However, a NEL collector requires significantly fewer resources than necessary to run the monitored service. In particular, NEL collectors typically do not need to make the same latency guarantees as interactive web requests. Therefore, a service's collection infrastructure is unlikely to have the necessary capacity for an operator to serve affected users from the collectors when these users are unable to access the service normally.

Load balancing and failover. NEL enables service operators to define arbitrary load balancing and failover behavior for their collectors. Inspired by the DNS `SRV` record [17], a NEL policy can specify an optional weight and priority for each collector. When choosing which collector to upload a report to, a user agent will randomly select a collector from those with the smallest priority value, weighted by their weight values. The user agent keeps track of whether uploads to a particular collector fail too frequently; if so, that collector is marked as *pending*, and taken out of rotation. This ensures that collectors with higher priority are only attempted if *all* collectors with lower priority have failed. This mechanism gives service operators maximum flexibility in determining how to configure their collection infrastructure.

If *all* of the collectors are unreachable, the user agent will retain the reports in memory for a small amount of time (typically 15 minutes). When this happens, it often indicates a complete loss of connectivity on the part of the user. During this time, the user agent will continue attempting to deliver the reports (typically once per minute, with exponential backoffs). If the reports have still not been delivered after several attempts, they are dropped. The short interval ensures that we have visibility into temporary connectivity losses, while not requiring much storage in the user agent.

Note that a NEL user agent will also upload reports summarizing the success or failure of its attempts to upload a NEL report to a collector; after all, attempts to upload NEL reports are also HTTP requests. We refer to these as *meta reports*. Such meta reports help a service provider detect problems that clients face in contacting its collectors. To prevent infinite recursion, user agents generate meta reports only for uploads of NEL reports that are not meta reports.

[4]Note that all of these must be different for the client to use a completely different set of nameservers to resolve the collector's hostname.

Feature	Domain Reliability	NEL W3C standard
Adoption model	Opt-in	Opt-out
Activation time	Browser start	After first successful request to an origin
Activation overhead	None	HTTP response headers
Coverage	Google origins	Any HTTPS origin

Table 3: Comparison of the two implementations of NEL: the version that monitors reachability from Chrome's users to Google's services versus the W3C standard. The latter incurs additional overhead to be generic.

Censorship resistance. Although we have found NEL useful in detecting and investigating state-sponsored censorship, NEL itself is not particularly resistant to censorship. An attacker who can block access to an origin can also trivially enumerate all NEL collectors for the origin, and block access to those collectors. Service operators could try to introduce a modicum of censorship resistance by hosting NEL collectors in cloud providers, thereby tying the fate of report collection to the cloud provider as a whole. An operator could also make it harder for an adversary to identify all of its collectors by returning different subsets of collectors to different clients [34]; however, a NEL collector may be discernable via network traffic analysis [14].

Report authenticity. Since NEL reports are generated by user agents running on untrusted client devices, there is nothing preventing clients from generating and uploading fraudulent reports. A service provider can account for this challenge by only reacting to problems that affect a large enough population of unique client IP addresses (typically dozens). This measure ensures that a malicious entity can only cause a service operator to react to fake outages if they control a large number of client devices (*e.g.*, a botnet operator). However, making NEL robust to fraud in this manner comes with the risk of minimizing the impact of an outage which affects a few IP addresses shared by many client devices, *e.g.*, when many devices are behind a shared NAT.

3.6 Domain Reliability

Thus far, this section has described the public standard [11] which is usable by all services and browsers, and available in Chrome as of version 69. We also implemented these ideas in an earlier proof-of-concept called *Domain Reliability*, which could only monitor reachability from Chrome users to Google services. There are some important differences between Domain Reliability and NEL (see Table 3).

- Since Domain Reliability was not generally available to all service operators and only monitored Google properties, it required users to explicitly opt in to report collection. With NEL, any user agent which implements the standard collects reports by default for origins which include NEL policies in their responses. A user can opt out of NEL either globally or on a per-origin basis.
- One consequence of NEL being opt-out is that its users will represent a more uniform sample of a service's userbase. Because of this, we expect to more confidently generalize results from NEL to non-NEL clients.
- With Domain Reliability, the list of origins for which clients generate reachability reports and the list of collectors to which they upload these reports was hard-coded into Chrome. Any updates to these lists were delivered as part of Chrome's regular update process, resulting in a multiple-week lead time to push any monitoring changes to our users. In contrast, with NEL, any web service gets to bootstrap the origins to monitor and the collector domains by including this information as headers in the service's HTTP responses. This allows monitoring changes to be picked up immediately, with the cost of increased overhead in client-server traffic.
- An implication of the previous point is that NEL can enable a client to upload reachability reports for a particular origin only after that client has successfully communicated with that origin at least once. Without doing so, the client would neither know that it must generate and upload reports for this origin, nor know which collectors to upload these reports to. Because its configuration was hard-coded in Chrome, Domain Reliability enabled monitoring of a client's reachability to Google's services even if that client had never successfully been able to communicate with any Google origin.
- Since Domain Reliability was only implemented in Chrome, it could not reveal reachability issues faced by users of other browsers. With NEL, a service can collect reports from any HTTP client that implements the proposed W3C standard [11].
- Because Domain Reliability's configuration was encoded in source code, we relied on the existing code review process to ensure that the configuration adhered to our desired security and privacy properties. Because this configuration was restricted (by policy) to Google properties, we did not have to downgrade Domain Reliability reports like is needed for NEL in some situations; instead, we ensured that Domain Reliability's hard-coded policy prevented reports from being sent at all in those situations.

4 Deployment Experiences

This section relays some of our experiences with the techniques described in the previous section. In each case, data alerted us to the presence of a problem and hinted at a cause based on the population of users reporting that problem (*i.e.*, the "shape" of the problem); however, these techniques are most useful in concert with other network diagnostic tools that can dig deeper into specific problems and provide "smoking gun" evidence of a particular cause.

Note that this section deals with Domain Reliability, the initial prototype of these concepts that we developed to mon-

```
$ traceroute X.Y.Z.33
                                                 Loss%   Snt   Last   Avg  Best  Wrst StDev
   1.|-- ip1.isp.net                              0.0%   100    0.2   1.1   0.2  49.3   5.5
   2.|-- ip5.isp.net                              0.0%   100    6.1   8.3   4.5  12.4   2.3
   3.|-- ip6.isp.net                              0.0%   100    8.4   8.8   7.5  36.5   3.9
   4.|-- ip7.isp.net                              0.0%   100    7.6   9.2   7.6  18.2   2.8
   5.|-- ip8.isp.net                             99.0%   100  2671. 2671. 2671. 2671.   0.0
   6.|-- ???                                     100.0   100    0.0   0.0   0.0   0.0   0.0
   7.|-- ???                                     100.0   100    0.0   0.0   0.0   0.0   0.0
   8.|-- ip9.isp.net                             99.0%   100  7314. 7314. 7314. 7314.   0.0
   9.|-- ???                                     100.0   100    0.0   0.0   0.0   0.0   0.0
  10.|-- ???                                     100.0   100    0.0   0.0   0.0   0.0   0.0
  11.|-- ip10.isp.net                            99.0%   100  5179. 5179. 5179. 5179.   0.0
  12.|-- ???                                     100.0   100    0.0   0.0   0.0   0.0   0.0
  13.|-- ip11.isp.net                            99.0%   100  2722. 2722. 2722. 2722.   0.0
  14.|-- ???                                     100.0   100    0.0   0.0   0.0   0.0   0.0
```

(a)

```
$ traceroute X.Y.Z.32
                                    Loss%   Snt   Last   Avg  Best  Wrst StDev
  1.|-- ip1.isp.net                  0.0%    10    0.2   0.2   0.2   0.3   0.0
  2.|-- ip2.isp.net                  0.0%    10    6.9   8.4   4.8  11.8   2.4
  3.|-- ip3.isp.net                  0.0%    10    7.7   8.3   7.5  10.7   1.1
  4.|-- ip4.isp.net                  0.0%    10   33.5  10.7   7.5  33.5   8.1
  5.|-- ip1.google.com               0.0%    10   49.7  49.2  43.3  55.2   4.4
  6.|-- ip2.google.com               0.0%    10   49.0  48.5  44.3  51.6   2.6
  7.|-- ip3.google.com               0.0%    10   47.6  47.8  44.3  53.2   2.9
  8.|-- X.Y.Z.32                     0.0%    10   48.9  49.8  45.9  58.7   4.6
```

(b)

Figure 4: (a) Traceroute to the affected IP address appears to show a routing loop in the last-mile ISP, and (b) traceroute from the same vantage point to an adjacent IP address exits the ISP within a few hops. Hostnames and IP addresses are anonymized.

itor traffic only to Google's services. We have monitored Google's services with it since Chrome 38 in 2014 and are currently migrating our monitoring to use NEL. Although Domain Reliability and NEL are not qualitatively different, Section 3.6 explains how one might expect our experiences to differ once we fully migrate to NEL.

4.1 Unreachability of a single IP address

In December of 2018, Chrome clients started reporting failures of TCP connections and QUIC sessions made to a single Google IP address. The problem affected all requests to that IP from all users in every ISP in one country for the following two weeks.

Further manual investigation revealed that traceroutes from an affected host to that IP were failing inside a transit ISP which dominates wired connectivity within that country. We also occasionally noticed IPs belonging to this transit provider in high-TTL hops of the traceroute, suggesting that packets were stuck in a routing loop in that ISP's network (see Figure 4). The problem was eventually resolved after we contacted the ISP, although we never learned how they presumably fixed it.

NEL was useful in this case in several ways.

- It let us quickly detect a problem we may have never noticed otherwise. The impacted IP served content that is visible to users but is not critical (*e.g.*, thumbnail images), which may be why we received no reports about this problem on social media, and why there was no visibility on crowdsourced sites like downdetector.com.
- NEL was additionally helpful in confirming that the problem had gone away, particularly since the ISP never notified us that they fixed it.
- Diversity of our NEL collectors was trivially useful in this case; a collector hosted on *any* IP other than the one impacted could have successfully received report uploads.

Although other network monitoring tools could have caught this problem (*e.g.*, active probing of all serving IPs), it would have been difficult to assess the scale of users impacted or even to achieve dense enough probe coverage to know how many ISPs were affected. It would have been

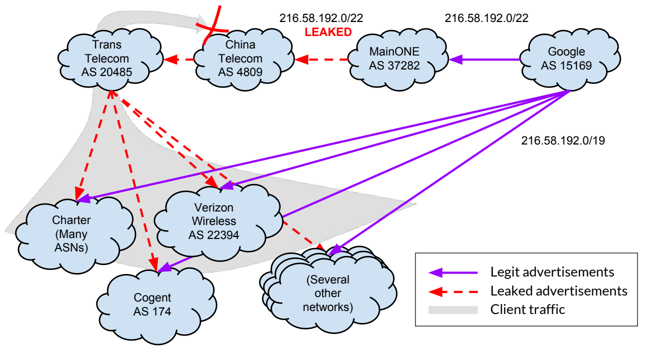


Figure 5: AS 37252 leaked prefixes to its peers, which ultimately misrouted traffic in several downstream ASNs. NEL quantified the impact of the leak on users in those downstream networks.

even more difficult to feel confident that the problem had been completely resolved.

However, NEL alone was not enough to diagnose this issue because it gave no information about the location of the problem other than the set of users affected. We needed other tools, like traceroute, to identify which network links were impacted and that most, if not all, of the Internet traffic in that country transits through one ISP.

4.2 BGP leakage

On November 12, 2018, AS 37282 leaked its routing table to its upstream providers. As shown in Figure 5, this leak rerouted traffic destined for some Google prefixes, causing packet loss for many of our users. The network operations community noticed this incident and the media widely reported on it.[5]

NEL saw this incident as an increase in connection timeouts for the leaked prefixes. Although many other monitoring tools had clear visibility of the leaked BGP routes [7], NEL directly told us the incident's impact on user traffic. In some networks, NEL reported that nearly all requests to these prefixes timed out. Moreover, diversity of our NEL collectors was useful in this case, because we had collectors running in IP prefixes not affected by the leakage.

[5]https://www.manrs.org/2018/11/route-leak-causes-major-google-outage/

On the other hand, NEL said nothing about the cause of the outage other than perhaps that the problem was localized to a specific set of prefixes. It would probably be most valuable to overlay NEL data on existing BGP leak detection tools, to distinguish relatively benign incidents of BGP leakage (*i.e.*, that do not impact much traffic) from major events like this one.

Note that BGP leaks that have no impact on the users of Google's services are likely since Google advertises and uses different IP prefixes to serve its users in different parts of the world. Hence, when an ISP erroneously advertises one of our prefixes, this has no impact on users if this ISP operates in a region far from where we use the affected prefix. NEL helps us avoid troubleshooting such inconsequential problems, whose impact would otherwise have been hard to determine only based on analysis of BGP data.

4.3 Malware-induced DNS hijacking

In February of 2018, NEL reported that users in a large ISP were resolving several of our domains to non-authoritative IP addresses belonging to a third-party cloud hosting provider. Clients could still complete requests to these domains (*i.e.*, most reports we received had type `ok`), suggesting the presence of a layer 3 proxy server. Although requests were successful, this was still concerning because a proxy could reduce performance and reliability. We could not find reports of the problem online or reproduce the issue ourselves from our CDN infrastructure in the ISP.

We later discovered a rogue DNS resolver associated with malware that was responding to DNS requests in much the same way. We theorize that either (1) an ISP DNS server had been compromised to resolve requests using the rogue resolver, or (2) many machines and/or home routers in that ISP had been similarly compromised.

This case highlights a key advantage of NEL over traditional active probing: NEL monitors actual user network conditions and configurations, which can differ from those of dedicated monitoring infrastructure. It can detect massive problems that are simply invisible to other forms of monitoring. However, this case also highlights that NEL is not necessarily very helpful in debugging problems; in this case, since NEL does not report which DNS resolver clients use, it did not help us make progress on the problem.

Note that collector diversity was unnecessary in this case because requests to these hijacked domains still worked. Nonetheless, NEL was helpful because it alerted us to the presence of a proxy server associated with malware.

4.4 Protocol-specific problems

On March, 17, 2017, NEL observed that users were having trouble connecting to Google services in two of our datacenters in the United States and Europe. On closer inspection, only clients using QUIC [23] were affected. This was corroborated by reports from users.[6]

Our operations team traced the problem back to a bad server configuration change, and mitigated it soon after. The problem caused machines in the affected datacenters to drop all traffic on established QUIC sessions. Although QUIC clients transparently fall back to TCP when QUIC cannot establish a connection, that did not help in this case because we only dropped packets *after* a connection was established.

This situation illustrates the value of black box traffic monitoring; if operations teams only monitor specific protocol-level metrics (*e.g.*, number of connections established), then there is a chance that those metrics do not tell the whole story. NEL lets operations teams know whether users' connections are healthy end-to-end.

NEL collector diversity was useful in this case because the problem was localized to a few datacenters; clients could successfully upload NEL reports to other locations.

4.5 Monitoring of NEL report uploads

In addition to discovering network outages, we have also leveraged NEL's collection infrastructure to monitor previously unmonitored infrastructure. Other operators of NEL collectors may do the same.

Domain Reliability monitors only Google's first-party services, but not customer-owned origins hosted on Google's cloud infrastructure. This is currently a blind spot in our monitoring. Moreover, NEL will not allow us to monitor customer-owned origins directly, since NEL's privacy design gives the *customer*, and not their cloud provider, control over whether reports are collected and where they are sent.

To help ameliorate this limitation, in addition to our existing diverse set of NEL collectors, we run another set of collectors hosted on our cloud infrastructure. As a result, whenever a user makes a request to a Google service, the user agent generates a NEL report and attempts to upload it to one of our cloud collectors with a probability based on the values of the weight fields in our NEL policy. The user agent then generates and uploads a *meta report* about the upload of the original report.

Although this technique does not grant us visibility into problems affecting individual cloud tenants (*e.g.*, a misconfigured tenant firewall), it at least lets us detect problems affecting our entire cloud infrastructure, even if those problems are localized to a small number of clients. For example, this helped us quickly confirm that the BGP leak in Section 4.2 also impacted our cloud infrastructure.

One caveat to meta reports is that they are not representative. Any sampling rates defined in the NEL policy are compounded for meta reports, making it more difficult to get a large enough collection of meta reports to derive a statistically meaningful signal. Clients with unreliable connectivity are more likely to attempt to send NEL reports, and to fail

[6]https://news.ycombinator.com/item?id=13892431

doing so. So, we see higher baseline error rates for NEL report traffic from such clients compared to the global set of NEL reports. As a result, we cannot compare NEL error rates for our cloud infrastructure and our non-cloud infrastructure; but, trend analysis on meta reports remains useful.

5 Deployment Challenges

This section discusses several practical challenges we encountered when deploying NEL. Other web service operators are likely to encounter similar hurdles.

5.1 Collector diversity

As seen in some of our case studies (§4), diversity in the deployment of collectors has been crucial to collect client unreachability reports in a timely manner. For example, in the BGP leak case, it was crucial that we had a collector in a different prefix from those leaked.

While such diversity existed in Google's infrastructure even prior to our deployment of NEL, hosting collectors across the globe, in multiple prefixes and AS numbers, and supporting multiple IP versions is unlikely to be straightforward for an arbitrary web service. Therefore, to ease the use of NEL by other web services, we envision public cloud providers offering "NEL collectors as a service." Like Google, other large cloud providers also have a rich diversity of global infrastructure that naturally lends itself for use as NEL collectors.

5.2 Overhead

NEL increases network traffic for both clients and service providers in two ways: 1) the additional header that the service provider must include in its responses to clients' HTTP requests, and 2) reports that clients upload to NEL collectors.

Response header overhead. As shown in Figure 3, servers must send two headers to activate NEL: NEL and Report-To. Although exact sizes vary depending on the number of collectors and the presence of non-default options, headers are typically several hundred bytes long and are uncompressed unless the client is using HTTP/2. Furthermore, there is currently no way for clients to tell the server whether they support NEL or already have activated NEL for an origin, so servers typically include headers on all requests to all clients. This could be particularly problematic when serving many small objects, in which case NEL headers could constitute a significant fraction of the total response size.

Service operators have several ways to curtail NEL's bandwidth usage. Operators could (1) only serve NEL on a fixed fraction of responses, (2) try to predict which clients support NEL based on their User-Agent header, or (3) only serve NEL headers on HTTP/2 connections, under the assumption that (due to NEL's relatively young age) all clients that support NEL also support HTTP/2.

Report upload overhead. NEL also incurs overhead whenever clients upload reports. Reports we receive from our clients are 532 ± 34 bytes long; clients batch these into uploads that contain an average of 1.3 reports each. Clients upload a batch of reports about an origin once per minute.

Clients pay additional bandwidth overhead for failed uploads: 1) Clients incur connection establishment overhead multiple times as they retry an upload to different collectors, and 2) each failed upload may itself generate a NEL report.

Service operators can control these overheads with sampling rates in the NEL header. In particular, because *most* requests succeed, the success_fraction field can have a large impact on total upload traffic. For example, over 90% of requests to our services succeed and reports about those requests do not contribute much to our ability to reason about network outages other than by establishing proper baselines. We set success_fraction to 0.05, but success reports are still almost 40% of our upload traffic.

5.3 Provisioning for bursty workloads

We could further reduce success_fraction, but at a cost. Although request failures are much rarer than successes, failures are very bursty. Major network outages can cause large networks to send tens or hundreds of times more NEL reports than they normally would. A service's NEL collection infrastructure must be provisioned to handle these cases or risk data collection failing at exactly when it is most needed.

NEL client retry logic compounds this by causing clients to retry uploads to collectors when they are overloaded. We currently mitigate this by having our collectors always return HTTP 200, which prevents clients from retrying uploads when the collection infrastructure is overloaded. If more explicit control is needed, the NEL specification requires user agents to stop sending reports to a collector entirely when that collector returns an HTTP 410 (*Gone*) response.

5.4 Application-layer retries

It can be tempting to compare error rates across different kinds of applications, domains, and URLs. For example, one might suspect that if one domain has four times the error rate of another then something must surely be wrong with the former domain. Although this *might* be true, application-layer retry logic could also explain the discrepancy.

For example, if a Web application makes a lot of AJAX requests to example.com and *retries those requests when they fail*, then the overall NEL error rate for example.com will appear higher. This is because successive request failures are likely not independent; if a request fails once, it is much more likely that a second request will also fail.

This logic also applies to user-initiated retries. If a particular request is more likely to elicit a retry (*e.g.*, a browser reload) from a user when it fails, that can also inflate the NEL error rate. For example, a user may be more likely to retry requests that are very important to them whereas they may simply abandon more trivial requests.

6 Future Work

End-to-end report encryption. As mentioned in Section 5.1, one easy way operators can increase the likelihood that clients can successfully upload NEL reports is to host collectors in cloud providers. However, NEL clients currently assume that operators trust collectors and therefore do not encrypt reports beyond uploading them using HTTPS. If an operator does not trust a cloud, their only option currently is to use the cloud as a layer 3 proxy and terminate HTTPS elsewhere. Future versions of NEL could encrypt reports end-to-end (*e.g.*, using PGP), which would decouple collectors (which would see reports as opaque blobs) and report analyzers (which could decrypt reports).

Automated triage. Individual NEL reports are rarely useful; we derive utility from examining collections of reports and identifying patterns in those collections. For example, if we see many timeouts of requests to many IPs in a prefix, that may indicate a problem with that entire prefix. We currently look for problems in a manually curated set of dimensions based on our past experience, but in the future could try to automatically identify problematic user populations without *a priori* knowledge of likely problems. Based on the "shape" of a given problem (*i.e.*, the distribution of different types of NEL reports), we could attempt to automate triage of the problem. For example, if we detect that all users in just one ISP are having difficulty accessing a domain, then we could automatically notify that ISP, since their network configuration is a likely culprit.

Reducing overhead. Future versions of NEL might add several mechanisms to reduce the overhead of NEL policy headers. For example, we might let services specify NEL policies in external URLs, using a mechanism like the proposed Origin Policy [33] or Site-Wide HTTP Headers [26] standards. By making their NEL policy object cacheable, service providers can preclude clients from having to fetch the policy from the external URL after every successful request. We may also let clients include request headers which indicate when the server should send NEL policy headers, using a mechanism similar to HTTP Client Hints [16]. This would prevent servers from needlessly sending NEL headers to clients that will ignore them.

7 Related Work

Earlier in Section 2, we reviewed prior work which shares NEL's aim of detecting and diagnosing service unreachability. Here, we compare NEL to other systems which also rely on client-side measurements.

There have been a number of previous client-side measurement systems focused on one of the following goals: continual collection of passive measurements [30], enabling users to measure their network [21], or orchestration of measurement campaigns [29, 25, 13]. All of these efforts aim to gather measurements with the aim of compiling performance distributions, characterizing middleboxes, measuring network topologies, etc. Since none of this data needs to be compiled in a timely manner, uploading via redundant paths has been unnecessary in these efforts, unlike in NEL. Moreover, since the measurements gathered in these systems do not contain application traffic, protecting user privacy has not been a concern.

Windows Error Reporting (WER) [15] is most similar in spirit to NEL in that it too uploads error reports gathered at the client. WER does have to pay attention to privacy by pruning crash reports before uploading them. However, since uploaded crash reports are analyzed at a later point in time [18], failover on the upload path was unnecessary in this case too.

Odin [10] enables Microsoft to gather measurements from their clients to their CDN. By incorporating active measurement logic into client applications, Odin preempts concerns regarding coverage associated with dedicated monitoring infrastructure. Like NEL, Odin too attempts to make report uploading fault-tolerant via proxies in third-party networks. Unlike NEL, since Odin only relies on measurements that it actively issues, purging reports to protect privacy is not a significant concern. Odin also cannot be used by services not managed by Microsoft.

8 Conclusion

Despite the wide range of solutions available today to detect and diagnose reachability issues over the Internet, service operators lack an important capability: the ability to quantify the number of clients affected by any particular outage. To fill this void, we presented NEL, which equips service providers with timely collection of reachability reports generated at the client. Incorporation of NEL's techniques into Chrome has proved invaluable over the last few years in monitoring reachability to Google's domains. Motivated by our experience, we have proposed NEL as a W3C standard to spur support for it in other user agents and to enable other service providers to benefit from this capability.

Acknowledgements

We thank the anonymous reviewers and our shepherd, KyoungSoo Park, for their valuable feedback. Many people at Google have contributed to this project, particularly through their experiences investigating problems NEL has detected. In particular, we thank Emma Christie, Fred Douglas, Logan Ingalls, Martijn Stevenson, Matthew Steele, Steve Wang, Wesley Darlington, and Yan Avlasov for investigating issues that NEL has detected over the years. Debashish Chatterjee, Francesco Marabotto, Jelmer Vernooij, and Mitchell Jeffrey have provided valuable support from Google's SRE organization. We thank Chris Bentzel for initial sponsorship of the project. Harsha Madhyastha's participation was funded in part by the National Science Foundation via award CNS-1563849.

References

[1] BGP errors are to blame for Monday's Twitter outage, not DDoS attacks. https://www.csoonline.com/article/3138934/security/bgp-errors-are-to-blame-for-monday-s-twitter-outage-not-ddos-attacks.html.

[2] A DNS hijacking wave is targeting companies at an almost unprecedented scale. https://arstechnica.com/information-technology/2019/01/a-dns-hijacking-wave-is-targeting-companies-at-an-almost-unprecedented-scale/.

[3] Dynatrace. http://www.dynatrace.com.

[4] ELK stack: Elasticsearch, Logstash, Kibana. https://www.elastic.co/elk-stack.

[5] Google Stackdriver. https://cloud.google.com/stackdriver/.

[6] RIPE Atlas. https://atlas.ripe.net.

[7] ThousandEyes. https://www.thousandeyes.com/solutions/bgp-and-route-monitoring.

[8] A. Barth. RFC 6454: The Web Origin Concept. *Internet Engineering Task Force (IETF)*, Dec. 2015.

[9] P. Anastas, W. R. Breen, Y. Cheng, A. Lieberman, and I. Mouline. Methods and apparatus for real user monitoring, 2010. US Patent 7,765,295.

[10] M. Calder, R. Gao, M. Schröder, R. Stewart, J. Padhye, R. Mahajan, G. Ananthanarayanan, and E. Katz-Bassett. Odin: Microsoft's scalable fault-tolerant CDN measurement system. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[11] D. Creager, I. Grigorik, A. Reitbauer, J. Tuttle, A. Jain, and J. Mann. Network Error Logging. W3C Editor's Draft, W3C Web Performance Working Group, 2019.

[12] A. Dainotti, C. Squarcella, E. Aben, K. C. Claffy, M. Chiesa, M. Russo, and A. Pescapé. Analysis of country-wide internet outages caused by censorship. In *ACM Internet Measurement Conference (IMC)*, 2011.

[13] M. Dhawan, J. Samuel, R. Teixeira, C. Kreibich, M. Allman, N. Weaver, and V. Paxson. Fathom: A browser-based network measurement platform. In *ACM Internet Measurement Conference (IMC)*, 2012.

[14] R. Ensafi, D. Fifield, P. Winter, N. Feamster, N. Weaver, and V. Paxson. Examining how the great firewall discovers hidden circumvention servers. In *ACM Internet Measurement Conference (IMC)*, 2015.

[15] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[16] I. Grigorik and Y. Weiss. HTTP Client Hints. Internet-Draft, HTTP Working Group, 2019. https://httpwg.org/http-extensions/client-hints.html.

[17] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). Technical report, IETF, 2000.

[18] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *International Conference on Software Engineering (ICSE)*, 2012.

[19] X. Hu and Z. M. Mao. Accurate real-time identification of IP prefix hijacking. In *IEEE Symposium on Security and Privacy*, 2007.

[20] E. Katz-Bassett, H. V. Madhyastha, J. P. John, A. Krishnamurthy, D. Wetherall, and T. E. Anderson. Studying black holes in the Internet with Hubble. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[21] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the edge network. In *ACM Internet Measurement Conference (IMC)*, 2010.

[22] M. Lad, D. Massey, D. Pei, Y. Wu, B. Zhang, and L. Zhang. PHAS: A prefix hijack alert system. In *USENIX Security symposium*, 2006.

[23] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, and J. Bailey. The QUIC transport protocol: Design and Internet-scale deployment. In *ACM SIGCOMM*, 2017.

[24] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *ACM SIGCOMM*, 2002.

[25] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao. Mobilyzer: An open platform for controllable mobile network measurements. In *MobiSys*, 2015.

[26] M. Nottingham. Site-Wide HTTP Headers. Internet-Draft, IETF Network Working Group, 2017. https://mnot.github.io/I-D/site-wide-headers/.

[27] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[28] P. Richter, R. Padmanabhan, N. Spring, A. Berger, and D. Clark. Advancing the art of Internet edge outage detection. In *ACM Internet Measurement Conference (IMC)*, 2018.

[29] M. A. Sánchez, J. S. Otto, Z. S. Bischof, D. R. Choffnes, F. E. Bustamante, B. Krishnamurthy, and W. Willinger. Dasu: Pushing experiments to the Internets edge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[30] S. Sundaresan, S. Burnett, N. Feamster, and W. De Donato. BISmark: A testbed for deploying measurements and applications in broadband access networks. In *USENIX ATC*, 2014.

[31] The Apache Software Foundation. Apache log files. https://httpd.apache.org/docs/2.4/logs.html.

[32] A. van Kesteren. Cross-origin resource sharing. Tech-

nical report, W3C, 2014.

[33] M. West. Origin Policy. Draft Community Group Report, Web Incubator Community Group, May 2017. https://wicg.github.io/origin-policy/.

[34] M. Zamani, J. Saia, and J. R. Crandall. TorBricks: Blocking-resistant Tor bridge distribution. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2017.

[35] Z. Zhang, Y. Zhang, Y. C. Hu, Z. M. Mao, and R. Bush. iSPY: Detecting IP prefix hijacking on my own. *IEEE/ACM Transactions on Networking*, 18(6):1815–1828, 2010.

Finding Network Misconfigurations by Automatic Template Inference

Siva Kesava Reddy Kakarla[1], Alan Tang[1], Ryan Beckett[2], Karthick Jayaraman[3], Todd Millstein[1,4], Yuval Tamir[1], and George Varghese[1]

[1]University of California, Los Angeles, USA. [2]Microsoft Research, Redmond, USA.
[3]Microsoft Azure, Microsoft, Redmond, USA. [4]Intentionet, Seattle, USA.

Abstract

Network verification to detect router configuration errors typically requires an explicit correctness *specification*. Unfortunately, specifications often either do not exist, are incomplete, or are written informally in English. We describe an approach to infer likely network configuration errors without a specification through a form of automated *outlier detection*. Unlike prior techniques, our approach can identify outliers even for complex, structured configuration elements that have a variety of intentional differences across nodes, like access-control lists, prefix lists, and route policies.

Given a collection of configuration elements, our algorithm automatically infers a set of parameterized *templates*, modeling the (likely) intentional differences as variations within a template while modeling the (likely) erroneous differences as variations across templates. We have implemented our algorithm, which we call *structured generalization*, in a tool called SELFSTARTER and used it to automatically identify configuration outliers in a collection of datacenter networks from a large cloud provider, the wide-area network from the same cloud provider, and the campus network of a large university. SELFSTARTER found misconfigurations in all three networks, including 43 previously unknown bugs, and is in the process of adoption in the configuration management system of a major cloud provider.

1 Introduction

Router configuration errors are a major cause of network outages [1, 17, 19, 30, 35, 37]. Accordingly, researchers have developed a variety of techniques to automatically identify such errors and/or to prove their absence. Techniques that began in academic research [16, 23, 24, 29] have migrated to industry via cloud vendors [2, 20], router vendors [3], and startups [4]. However, these approaches have an important practical limitation: users must provide an explicit, formal *specification* of the network's intended behaviors (e.g., reachability requirements) [14, 18, 23, 29]. In practice, such specifications often do not exist, and when they do exist, they tend to be informal, incomplete, and ambiguous. A few tools do not require a specification [15, 16] but then are limited to identifying *generic* configuration errors (e.g., forwarding loops, duplicate IP addresses), independent of the network's particular policy intent.

In this paper, we develop an approach to identify *network-specific* misconfigurations without a specification, through a form of outlier detection. The *bugs as outliers* paradigm [13] is natural for network configurations, since, by design, many nodes' configurations are intended to be highly similar to one another (e.g., all nodes playing the same *role* in the network). We refer to any logical unit of a configuration, such as a BGP session configuration or access-control list, as a *segment*. Given a set of configuration segments that are intended to be similar, our goal is to automatically identify likely misconfigurations and provide actionable feedback to fix them.

Prior work in outlier detection (§ 8) for network configurations [12, 28] assumes that configuration segments are intended to be *exactly equivalent* to one another. Such approaches can identify outliers in the simpler aspects of a configuration, such as the set of DNS servers and the MTU values on interfaces. However, exact equivalence is much too strong an assumption for detecting useful outliers for complex configuration segments like access-control lists and route policies. Such segments often have a variety of intentional policy differences across nodes (e.g., the treatment of local hosts or services). Hence, outlier detection must be able to distinguish intentional differences from ones that represent likely errors.

We describe a general approach to outlier detection that addresses this limitation. Given a set of configuration segments, our algorithm, which we call *structured generalization*, infers one or more segment *templates*. Each template is a segment definition that is optionally parameterized by various pieces of data (e.g., the IP address used on a particular line). These templates serve as a compact summary of the differences across network policies and induce an equivalence relation on the nodes: two nodes are considered "equivalent" if they are instantiations of the same template. In other words, we model the (likely) intentional differences between nodes as variations *within a template* while modeling the (likely) erroneous differences as variations *across templates*.

The key challenge is to infer templates that result in useful equivalence relations. This requires a delicate balance between templates that cover too few or too many configurations. Parameterization is necessary to account for intentional differences between segments. However, supporting arbitrary parameterization would lead to overly-general templates that mask important differences. Similarly, the ordering of some configuration lines (e.g., consecutive `permit` lines in an ACL) is semantically transparent. Hence, a single template should admit such reordering. However, a single template that admits arbitrary reordering (e.g., arbitrary ACL `permit` and `deny` lines) may result in masking semantic differences, thus potentially hiding critical configuration errors.

Our structured generalization meets this challenge through a novel two-level approach to matching segments with one another. *Sequence alignment* is used to align *blocks* of segment lines with one another, thereby admitting insertions and deletions but ensuring that the order of blocks is respected. However, the similarity measure for this alignment employs *bipartite matching* to match the lines within the blocks, thereby admitting line reordering and also naturally inducing template parameters wherever two matched lines differ. Our algorithm is parameterized by the definition of blocks and the cost function for bipartite matching, which respectively control the amount of reordering and parameterization. We also provide instantiations of this generic algorithm for access-control lists, prefix lists, and route policies.

We have implemented structured generalization in a tool called SELFSTARTER. We applied SELFSTARTER to identify configuration outliers in three different kinds of networks: a large number of datacenters in a cloud provider network, totaling on the order of $O(10000)$ routers; the wide area network (WAN) of a large cloud provider, containing hundreds of routers; and the department routers of a large university campus network, containing $\sim$100 routers. In these networks, we applied SELFSTARTER to three heavily-used segment types: *access-control lists* (ACLs), which contain a sequence of `permit` and `deny` lines that determine which packets to accept; *prefix lists*, which have a similar structure as ACLs and are used to determine which route announcements to accept during routing; and *route policies*, which flexibly match sets of route announcements and update them in various ways (e.g., to add a community tag).

For the datacenter, SELFSTARTER identified 1168 outliers, of which 630 were investigated and all determined to be true positives. For the wide area, SELFSTARTER identified 56 route policy outliers, of which 33 were investigated and all were determined to be true positives. As SELFSTARTER found new bugs that were previously unknown to operators, it is in the process of being adopted in the configuration management process of the WAN. However, SELFSTARTER was much less successful in identifying true positives for prefix lists in the wide area; the reasons are explained in §6. For the university network, SELFSTARTER identified 6 ACL outliers, of which 3 were investigated and all were determined to be true positives. SELFSTARTER's metatemplates made it easy for the network operators to quickly classify outliers as true/-false positives and to remediate the actual misconfigurations. Further, the templates that SELFSTARTER generated closely matched any existing "golden" templates or configurations for these networks.

We make the following contributions:

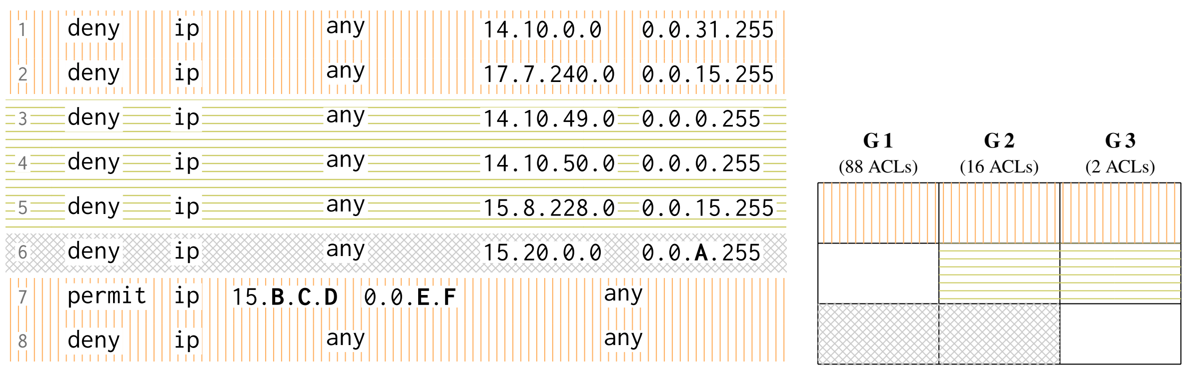
1. **Automatic Template Inference:** To our knowledge, we are the first to propose the idea of automatic template inference for network configuration segments and to employ it to identify network misconfigurations (§2).
2. **Structured Generalization:** We present a novel algorithm for automatic inference of parameterized templates for network configuration segments that combines sequence alignment and bipartite matching in a two-level structure to support controlled forms of parameterization and reordering (§3 and §4).
3. **Implementation:** We have implemented structured generalization in a practical tool called SELFSTARTER (§5).
4. **Evaluation:** We describe our empirical evaluation of SELFSTARTER on several large real-world production networks, demonstrating that in most cases it generates high-quality outliers with a low false positive rate (§6).

2 Using SELFSTARTER

We describe an example of SELFSTARTER's output and its use in finding misconfigurations in the campus network of a large university. SELFSTARTER was given as input the configurations of 106 *building routers* (one of the roles in the network), along with a regular expression capturing the name(s) of an access-control list (ACL) of interest used on the routers.[1] Given this input, SELFSTARTER automatically inferred three templates, each of which is an ACL definition that is parameterized by zero or more parameters.

Figure 1 shows SELFSTARTER's output for this example. Figure 1(a) shows a *metatemplate*, which is a concise representation of the three inferred templates, highlighting their commonalities and differences. A metatemplate is a sequence of parameterized configuration lines. Capital letters like `A` and `B` are *parameters*, representing values that differ across ACLs that contain the corresponding line. The metatemplate is *complete*: every line that appears in some ACL is represented by a line in the metatemplate.

[1]We allow regular expressions instead of simple strings since some organizations append metadata to ACL names, so two ACLs with slightly different names may still be intended to be similar.



```
1  deny    ip              any                14.10.0.0     0.0.31.255
2  deny    ip              any                17.7.240.0    0.0.15.255
3  deny    ip              any                14.10.49.0    0.0.0.255
4  deny    ip              any                14.10.50.0    0.0.0.255
5  deny    ip              any                15.8.228.0    0.0.15.255
6  deny    ip              any                15.20.0.0     0.0.A.255
7  permit  ip  15.B.C.D  0.0.E.F                       any
8  deny    ip              any                          any
```

(a) ACL Metatemplate

(b) Groups

Figure 1: SELFSTARTER output for an ACL Regex in a university network. Groups 2 and 3 are confirmed to be anomalous.

Figure 1(b) identifies the three templates that SELFSTARTER inferred. Each column represents a *group* of nodes that share a common template. Colors allow users to easily see which lines of the metatemplate belong to each template. For example, the template for the 88 ACLs in Group 1 contains the metatemplate lines that are colored orange (vertical bars) and gray (crosshatching), i.e., lines 1, 2, 6, 7, and 8. Similarly, the template for Group 2 consists of lines 1-8, and the template for Group 3 consists of lines 1-5, 7, and 8.

Though each of the example ACLs contains at most eight lines, manually scanning the configurations from all 106 routers to find outliers would be onerous and error-prone. Furthermore, partitioning the ACLs based on *exact equivalence* would result in 53 different groups, since each building has two routers with an identical ACL, but parameter values differ from building to building. In contrast, SELFSTARTER's output makes it easy for network engineers to identify outliers and to understand exactly how they differ from non-outliers. Specifically, SELFSTARTER helps engineers to identify two types of outliers, which we now describe.

Group Outliers. If SELFSTARTER produces multiple groups, the network engineer can compare the templates for these groups to decide whether one or more groups are misconfigured. Groups that are relatively small in size are particularly likely to be the results of misconfiguration. For example, in Figure 1(b) the vast majority of the ACLs are in Group 1, indicating that Groups 2 and 3 are suspect.

In fact, for this example, the network engineers have confirmed that Groups 2 and 3 represent misconfigurations. Group 3 erroneously omits line 6, allowing some flows that should be denied. Further, both Groups 2 and 3 erroneously include lines 3-5. While these lines used to be required to prevent access to certain infrastructure servers, those servers were phased out and the lines were supposed to be removed. Further, some of the denied IP addresses had since been reassigned to servers that are intended to be accessible. Hence these lines deny some flows that should be allowed.

Parameter Outliers. Structured configuration segments, like ACLs, often differ across nodes. Hence, the ability to parameterize is critical for generating templates that identify similarities without requiring exact equivalence. For each parameter, SELFSTARTER maintains a mapping from nodes to parameter values for the engineer to inspect.

A parameter error is present if some field in a line is supposed to be constant across all the routers, but it is not. SELFSTARTER guarantees that a parameter in some field of a line in the metatemplate indicates that there must be at least two different values for that field across the given configuration segments. Thus, a parameter error is identified in the metatemplate by a field that contains a parameter instead of a constant. In our example, out of the 104 ACLs that contain line 6, 94 use the mask 255 for parameter A, while 10 of them use the mask 127. The network engineers confirmed that the 10 ACLs are erroneous, permitting more traffic than intended.

SELFSTARTER is useful for finding errors and inconsistencies in networks that are managed through manual creation of individual node configurations. Perhaps surprisingly, we have also found SELFSTARTER to be useful for networks that employ forms of automation to manage configurations.

First, many network engineers employ *configuration templates*, with one parameterized template per role in the network. This simplifies network management since node configurations can be created by instantiating the relevant template with node-specific parameter values. For example, the university network described above has a template for all building routers. However, this network still suffered from multiple misconfigurations identified by SELFSTARTER. The problem is that node configurations tend to drift over time from their original templates. Such *template drift* happens for several reasons. First, operators often must manually edit a node's configuration, for example to quickly address a problem or to perform maintenance of various kinds. Second, the templates themselves get updated over time, and often operators are relied upon to manually perform the necessary configuration updates. Hence there is typically still considerable manual configuration in practice, which can easily lead to errors.

Second, some networks employ automated scripts to deploy, update, and validate configurations consistently. In this situation, SELFSTARTER is useful to protect against bugs in the automation software itself. Automation may actually *increase* the need for validation tools like SELFSTARTER, since small errors in the automation can lead to large, network-wide misconfigurations. The wide-area network (WAN) that we analyzed employs automation scripts for various purposes, and SELFSTARTER discovered misconfigurations that were due to previously unknown script errors. Once reported, the network engineers promptly confirmed and fixed the errors.

3 Structured Generalization

Our *structured generalization* algorithm takes as input a collection of configuration segments and outputs a metatemplate for these segments in the form shown in Figure 1. After an overview of the key challenges, we present the generic algorithm and then describe how it is instantiated for ACLs, prefix lists, and route policies.

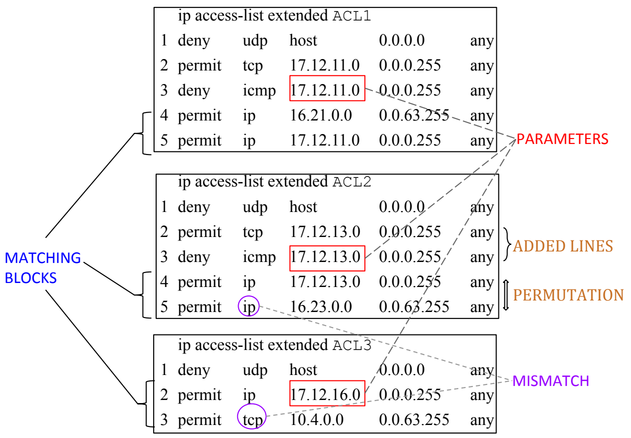
To provide high-quality outliers, the algorithm must be given configuration segments that are intended to be similarly structured. For example, it is common for routers to be partitioned into *roles* (e.g., border routers, core routers) and for the routers within a role to be configured similarly. We rely on the user to provide an appropriate set of configuration segments to be templated. In our experience (see §6), operators know the roles in their network and can quickly identify the segments that are intended to be similar, so this requirement does not pose a large burden in practice.

3.1 Challenges

Suppose that we wish to create a metatemplate for the three different configurations of the same ACL shown in Figure 2. We use this example to illustrate the challenges that our algorithm must address.

Consider the first two ACLs in Figure 2. Their first lines are identical, so clearly they should be matched to one another. However, their second lines differ — they apply to different source IP addresses. A naive approach is to simply not match these lines to one another. However, such an exact-matching criterion is much too strong, as it is common for corresponding ACLs to be similar but not identical across nodes, for example to treat local addresses specially. Therefore, the algorithm must be able to match non-identical lines to one another. This requirement is met using *parameterization*. In this example, we can introduce a parameter to represent the third octet in the source IP address, indicating that this octet differs in each ACL while the rest of the line is identical.

While limited parameterization is necessary, arbitrary parameterization would yield undesirable results. For example, it would not be useful if the metatemplate matches a `permit`


```
  ip access-list extended ACL1
1 deny    udp   host       0.0.0.0     any
2 permit  tcp   17.12.11.0 0.0.0.255   any
3 deny    icmp  17.12.11.0 0.0.0.255   any
4 permit  ip    16.21.0.0  0.0.63.255  any
5 permit  ip    17.12.11.0 0.0.0.255   any

  ip access-list extended ACL2
1 deny    udp   host       0.0.0.0     any
2 permit  tcp   17.12.13.0 0.0.0.255   any
3 deny    icmp  17.12.13.0 0.0.0.255   any
4 permit  ip    17.12.13.0 0.0.0.255   any
5 permit  ip    16.23.0.0  0.0.63.255  any

  ip access-list extended ACL3
1 deny    udp   host       0.0.0.0     any
2 permit  ip    17.12.16.0 0.0.0.255   any
3 permit  tcp   10.4.0.0   0.0.63.255  any
```

Figure 2: Configurations of the ACLs matching the `ACL*` regex from three different routers highlighting the challenges.

line with a `deny` line, as their behavior is not at all similar. As a more subtle example, consider the last line in `ACL3` in Figure 2. This line is similar to the last line in `ACL2`. However, since the two lines specify different protocols (tcp vs. ip), it is unlikely that they are intended to serve the same function in the ACLs. Hence, despite being similar, it may not make sense to match these two lines. Therefore, there is a need for an ability to specify different constraints on what can and cannot be parameterized for different segment types.

Now consider lines 4 and 5 in `ACL1` and `ACL2`. Line 4 in `ACL1` is similar to line 5 in `ACL2`. To match these lines to one another, the algorithm must support *reordering* of configuration lines. Doing so requires a scoring metric to determine which line pairs are the best matches. Such a scoring metric also naturally handles missing lines, as in `ACL3`, resulting in there being multiple options for how to match the lines present in `ACL3` to those in the other ACLs.

As with parameterization, allowing arbitrary line reordering would yield undesirable results. Specifically, reordering a `permit` line and a `deny` line can, in general, change the semantics of an ACL. Therefore, matching lines across ACLs in a way that requires such a reordering can potentially mask important differences between ACLs. Thus, as with parameterization, we need the ability to specify constraints on the allowable reorderings for different segment types.

Our structured generalization algorithm, described below, meets these challenges through a novel two-level approach.

3.2 Algorithm

The algorithm partitions segments into *blocks* of lines that must not be reordered with one another and hence are matched using sequence alignment. However, when matching two blocks with one another, the algorithm employs bipartite matching on their lines, thereby supporting line reordering within blocks and also inducing parameters wherever two

Algorithm 1 STRUCTURED GENERALIZATION

```
Input: $S_1, S_2, \ldots S_n$ - Sequence of segments to be templated
Output: Metatemplate for $S_1, S_2, \ldots S_n$
1: Metatemplate $T \leftarrow S_1$
2: for $S_i \leftarrow \{S_2, \ldots, S_n\}$ do
3:     Block sequence $B_1 \leftarrow$ GETBLOCKSEQUENCE($T$)
4:     Block sequence $B_2 \leftarrow$ GETBLOCKSEQUENCE($S_i$)
5:     Alignment $A \leftarrow$ ALIGNSEQUENCES($B_1$, $B_2$, MISMATCHSCORE)
6:     $T \leftarrow$ GENERATEMETATEMPLATE($A$, $B_1$, $B_2$)
7: end for
8: $T \leftarrow$ MINIMIZEPARAMETERS($T$)
9: return $T$
```

matched lines differ. The amount of reordering and parameterization are respectively controlled through a function that partitions segments into blocks and a function that determines edge weights for bipartite matching.

The structured generalization algorithm is shown in Algorithm 1. The algorithm starts by treating the first segment S_1 as the initial metatemplate (Line 1). It then iteratively combines the current metatemplate with each remaining configuration segment, one at a time, to produce the final metatemplate.

At each iteration the algorithm performs three main steps:

1. Block Generation: A *block* is a contiguous sequence of lines within a segment that can be reordered with one another. The GETBLOCKSEQUENCE function partitions each segment into blocks (Lines 3 to 4) and is specific to a particular segment type. For example, for ACLs the GETBLOCKSEQUENCE function partitions a segment into maximal sequences of lines that have the same action (`permit` or `deny`). The block abstraction and corresponding GETBLOCKSEQUENCE function provide a natural way to admit reorderings within a block while forbidding those across blocks.

2. Block Alignment and Line Matching: Blocks are matched in the two block sequences B_1 and B_2. To prevent cross-block reorderings, a standard *sequence alignment* algorithm is employed at the block level (Line 5). Such an algorithm finds a minimum-cost alignment between the two block sequences, allowing gaps but not reorderings.

To meet the need to support within-block reorderings, our algorithm leverages the fact that the sequence alignment algorithm requires a function MISMATCHSCORE that provides the cost of matching two blocks to one another.[2] The MISMATCHSCORE function is shown in Algorithm 2. The function first breaks the two blocks into lines, and then it performs a minimum-weight bipartite matching between the two line sequences using a standard matching algorithm, thereby supporting line reordering. The function returns the score for the best matching and also the matching itself.

The MISMATCHSCORE function uses another function, LINESCORE, to assign weights to each edge in the bipartite graph (between a pair of lines). This function is specific

[2]Sequence alignment algorithms also require a function to provide the cost of introducing a gap; we use a simple metric for this cost based on the size of the unmatched block.

Algorithm 2 MISMATCHSCORE

```
Input: $\{b_1, b_2\}$ - Blocks to be matched
Output: A score for aligning $b_1, b_2$ and a matching between their lines
1: Line sequence $L_1 \leftarrow$ GETLINESEQUENCE($b_1$)
2: Line sequence $L_2 \leftarrow$ GETLINESEQUENCE($b_2$)
3: Score $S$, Matching $M \leftarrow$ MINIMUMWEIGHTBIPARTITEMATCHING($L_1$,
   $L_2$, LINESCORE)
4: return $S$, $M$
```

to each type of segment and provides a flexible way to prevent undesired line matchings and to prioritize among different possible line matchings.

3. Parameterization: Once the two block sequences have been aligned, the metatemplate is generated (Line 6). As described above, for each pair of blocks that are aligned with one another, the MISMATCHSCORE function also provides a matching at the line level. For each pair l_1 and l_2 of matched lines, the metatemplate includes their *least general generalization* (lgg) [34]. Intuitively, this is simply a version of l_1 where a parameter is introduced in each field where it differs from l_2. For example, the lgg of the second lines in the first two ACLs in Figure 1 is `permit tcp 17.12.P.0 0.0.0.255 any`, where `P` is a new parameter.

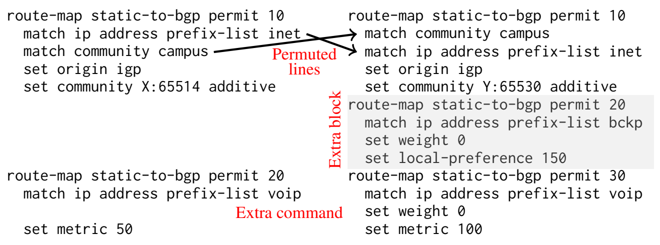
As a final step, the number of parameters in the metatemplate is globally minimized (Line 8). Specifically, let $P(S)$ denote the value that parameter P takes in segment S. If two parameters P_0 and P_1 in the metatemplate have the property that $P_0(S_i) = P_1(S_i)$ for each segment S_i, then the parameters are merged into a single parameter.

3.3 Instantiation for ACLs and Prefix Lists

This subsection discusses how the structured generalization algorithm is instantiated for ACLs and prefix lists, which are handled identically. Doing so requires providing specific GETBLOCKSEQUENCE and LINESCORE functions to the generic algorithm above. The goal is to take advantage of the particular properties of ACLs and prefix lists to both allow for flexible templating and ensure that generated templates are *actionable*, i.e., they facilitate the identification of common configuration errors.

The GETBLOCKSEQUENCE function partitions an ACL or prefix list into maximal sequences of lines that have the same action (`permit` or `deny`). This allows reorderings guaranteed to have no behavioral effect. The notion of blocks can be relaxed to admit more reorderings. For example, reordering a `permit` line with a `deny` line does not change behavior when the sets of packets that they handle are disjoint. However, in practice, we have not encountered the need to support such reorderings, so the extra complexity is not warranted.

The LINESCORE function for ACLs and prefix lists prohibits matching lines that differ in either their action (`permit` or `deny`) or protocol by returning an infinite weight. All other differences (e.g., in source or destination IP addresses) are



```
route-map static-to-bgp permit 10            route-map static-to-bgp permit 10
  match ip address prefix-list inet            match community campus
  match community campus                       match ip address prefix-list inet
  set origin igp                               set origin igp
  set community X:65514 additive               set community Y:65530 additive
                                             route-map static-to-bgp permit 20
                                               match ip address prefix-list bckp
                                               set weight 0
                                               set local-preference 150
route-map static-to-bgp permit 20            route-map static-to-bgp permit 30
  match ip address prefix-list voip            match ip address prefix-list voip
                                               set weight 0
  set metric 50                                set metric 100
```

Figure 3: Example: Two route policies.

allowed but are penalized by an increase in overall score. Specifically, consider some field of the two lines that is allowed to differ (e.g., the first octet of the source IP address). If the lines agree on the value of this field, then the overall score is unchanged. If one field contains a constant while the other contains a parameter name (which must have been introduced in an earlier iteration), then the overall score is increased by 1, since an exact match is preferable. Finally, if the lines contain different constant values in this field, then the overall score is increased by 2, since a new parameter must be introduced in order to match the lines, so either of the above two cases is preferable. Despite its simplicity, this scoring function works well across all of our experiments (§6).

§4 contains a detailed example of the structured generalization algorithm applied to the ACLs in Figure 2.

3.4 Instantiation for Route Policies

Route policies are flexible configuration segments used in several contexts, including route redistribution filtering, policy-based routing, and BGP policy implementation. A route policy is defined as a sequence of *route map clauses*, each of which contains an action (e.g., `permit`), a list of `match` lines, and a list of `set` lines.[3] The `match` lines provide a flexible way to select route announcements of interest based on the route attributes. The `set` lines then update matching announcements (e.g., to add a particular community tag). Figure 3 shows two example route policies.

The GETBLOCKSEQUENCE function for route policies puts each clause in one block, so the left and right route policies in Figure 3 respectively contain two and three blocks. This allows the `match` lines within a route map to be reordered with respect to one another, and similarly for the `set` lines.[4] Since all `match` lines have to succeed for a route announcement to match the route map, reordering them has no effect. In principle, reordering `set` lines can change behavior, specifically if one `set` line reads a value that was updated by a previous `set` line. In our experience, however, order dependence is rare: we have encountered only one such situation (where the deletion to a community attribute was followed by an addition). Therefore we opted to allow reordering of `set` lines within route maps to handle the common case properly.

The LINESCORE function for route policies allows two lines to be matched in the bipartite matching only if they refer to the same *attribute* of the route announcement. For example, the `match community campus` line in the left route policy of Figure 3 can only match against other `match community` lines, and the following `set origin igp` line can only match against other `set origin` lines.

With route policies it is common for `match` lines to refer to prefix lists and access lists defined elsewhere in the configuration. For example, the first `match` line in the left route policy refers to the prefix list named `inet`. When scoring two lines that refer to a named segment, we choose to perform a *shallow* comparison that considers them to exactly match if they refer to same-named segments. An alternative would be to perform a *deep* analysis that ignores the names and instead expands the definitions of these segments in order to recursively compare them. From our experiments on a large cloud provider network, we found that simple name comparison works well, since the network names segments identically across configurations. Further, if two same-named segments do differ in their definitions, that will be caught during metatemplating of those segments.

Figure 3 illustrates the best alignment of the two route policies, based on the GETBLOCKSEQUENCE and LINESCORE functions described above. Specifically, the second clause of the left policy is aligned with the third clause of the right policy. That is the lowest-cost alignment since these route maps match on the same-named prefix list announce and both have a line to set the `metric`.

4 A Templating Example

This section describes how Algorithms 1 and 2 generate a metatemplate for the three ACLs in Figure 2. Figure 4 shows `ACL1` and `ACL2` side by side; we use superscripts to uniquely refer to each `permit` and `deny` line. The STRUCTURED GENERALIZATION algorithm designates `ACL1` as the initial metatemplate (Line 1) and iteratively incorporates `ACL2` and `ACL3` to produce the final metatemplate (Lines 2 to 7).

In the first iteration, the "Block Generation" step partitions `ACL1` (Line 3) and `ACL2` (Line 4) into blocks. For `ACL1`, the algorithm generates the block sequence B_1 consisting of four blocks — $\mathsf{D_a}$, $\mathsf{P_a}$, $\mathsf{D_b}$, and $\mathsf{P_b}$ — where $\mathsf{D_a}$ contains deny^1, $\mathsf{P_a}$ contains permit^1, $\mathsf{D_b}$ contains deny^2, and $\mathsf{P_b}$ contains permit^2 and permit^3. The algorithm generates an analogous block sequence B_2 for `ACL2`, with four blocks that we denote $\mathsf{D_x}$, $\mathsf{P_x}$, $\mathsf{D_y}$ and $\mathsf{P_y}$.

Next, in the "Block Alignment and Line Matching" step, the algorithm uses the sequence alignment algorithm, which in turn relies on the MISMATCHSCORE(b_1, b_2) function in Algorithm 2. It turns out that the following alignment A is the

[3]Here we have used the syntax from Cisco IOS; the JunOS syntax from Juniper uses different keywords but is semantically similar.

[4]Technically it also allows `match` and `set` lines to be reordered with one another, but that is not syntactically legal so will never arise.

```
ip access-list extended ACL1
 deny^1   udp  host        0.0.0.0     any
 permit^1 tcp  17.12.11.0  0.0.0.255   any
 deny^2   icmp 17.12.11.0  0.0.0.255   any
 permit^2 ip   16.21.0.0   0.0.63.255  any
 permit^3 ip   17.12.11.0  0.0.0.255   any
```

```
ip access-list extended ACL2
 deny^3   udp  host        0.0.0.0     any
 permit^4 tcp  17.12.13.0  0.0.0.255   any
 deny^4   icmp 17.12.13.0  0.0.0.255   any
 permit^5 ip   17.12.13.0  0.0.0.255   any
 permit^6 ip   16.23.0.0   0.0.63.255  any
```

Figure 4: Side by side comparison of ACL1 and ACL2

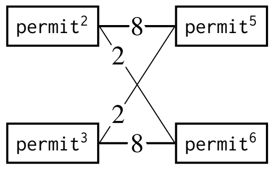


Figure 5: Bipartite graph of P_b and P_y

```
ip access-list extended ACL*
 deny         udp  host        0.0.0.0     any
 permit       tcp  17.12.A.0   0.0.0.255   any
 deny         icmp 17.12.B.0   0.0.0.255   any
 permit^3,5   ip   17.12.C.0   0.0.0.255   any
 permit^2,6   ip   16.D.0.0    0.0.63.255  any
```

```
ip access-list extended ACL3
 deny      udp  host        0.0.0.0     any



 permit^7  ip   17.12.16.0  0.0.0.255   any
 permit^8  tcp  10.4.0.0    0.0.63.255  any
```

Figure 6: Comparing the metatemplate for ACL1 and ACL2 with ACL3

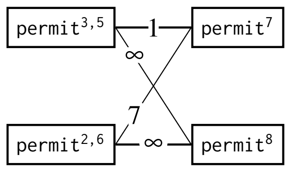


Figure 7: Bipartite graph of last permit block

optimal alignment for the two sequences.

$$(D_a, P_a, D_b, P_b)$$
$$(D_x, P_x, D_y, P_y)$$

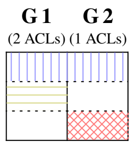
The single line in D_a is matched with the single line in D_x, and similarly for the next two pairs of blocks in the alignment. For the last pair of blocks (P_b, P_y), the algorithm constructs the bipartite graph shown in Figure 5 to determine line matchings. The LINESCORE(l_1, l_2) function assigns the edge weight of permit^2 and permit^6 as 2 since the lines differ in the second octet of the source IP, but the edge weight of permit^2 and permit^5 is assigned as 8 since they differ in four octets total across the source IP and source mask. Therefore, the minimum weight matching (Line 3) matches permit^2 with permit^6 and permit^3 and permit^5, thereby performing the allowable reordering.

The "Parameterization" step generates the metatemplate by introducing parameters based on the produced sequence alignment and line matchings (Line 6). The resultant metatemplate of ACL1 and ACL2 is shown on the left side in Figure 6. The two ACLs have a common template; the line produced by merging permit^3 with permit^5 is shown as permit^{3,5} in the metatemplate, and similarly for permit^{2,6}.

The next iteration incorporates ACL3. The metatemplate from the first iteration and ACL3 are both partitioned into blocks, and then these blocks are aligned in the same way as described above. The best block alignment and line matching are shown in Figure 6. As an example, the bipartite graph constructed to calculate the MISMATCHSCORE of the last block in each block sequence is shown in Figure 7. The edge weight of permit^{3,5} and permit^7 is 1 since there is already a parameter in the metatemplate for the source IP address octet where the two lines disagree. The LINESCORE function gives the edge between permit^{3,5} and permit^8 a score of ∞ since lines with different protocols cannot be matched. The minimal-weight matching matches permit^{3,5} with permit^7 and leaves lines (permit^{2,6} and permit^8) unmatched. This

```
ip access-list extended ACL*
 deny           udp   host        0.0.0.0     any
 permit         tcp   17.12.A.0   0.0.0.255   any
 deny           icmp  17.12.B.0   0.0.0.255   any
 permit^3,5,7   ip    17.12.C.0   0.0.0.255   any
 permit^2,6     ip    16.D.0.0    0.0.63.255  any
 permit^8       tcp   10.4.0.0    0.0.63.255  any
```

(a) Metatemplate of all three ACLs



(b) Groups

```
ACL1: [A = 11, B = 11, C = 11, D = 21]
ACL2: [A = 13, B = 13, C = 13, D = 23]
ACL3: [C = 16]
```

(c) Router parameter value map

Figure 8: Result of templating ACL1, ACL2 and ACL3

example demonstrates the need to match blocks but also to identify gaps due to missing/extra lines.

The resultant metatemplate for the three ACLs is shown in Figure 8(a), the ACL groups are shown in Figure 8(b), and the parameter-value map is shown in Figure 8(c). Finally, as explained in §3.2, the algorithm can create more parameters than necessary. For example, in Figure 8, parameters A, B and C are redundant: for every router *R* that requires all of these parameters, *R* instantiates the parameters with the same value. Therefore, the parameter minimization step (Line 8) merges A, B and C into a single parameter.[5]

5 Implementation

SELFSTARTER[6] takes as input a set of router configurations and a segment name or regular expression. It outputs a

[5]A degenerate case occurs when two parameters are never required together by any router; in that case we do not merge the parameters since they are likely to be logically unrelated.

[6]https://github.com/SivaKesava1/SelfStarter

metatemplate for all segments that match the given name or regular expression, a partitioning of each segment into groups that share a common template, and a parameter mapping for each segment, as shown in Figure 8.

SelfStarter is written in Python. It uses PyBatfish,[7] a client for the Batfish network configuration analysis tool [16], to parse the raw vendor-specific configuration files into Batfish's vendor-agnostic format. This format provides a structured representation of the configuration data, and our algorithm works directly on this representation. Batfish can parse many different configuration formats, including those from Cisco, Juniper, and Arista, so in turn SelfStarter can infer templates for segments from all of these vendors.

Our structured generalization algorithm uses a standard algorithm for sequence alignment based on dynamic programming, in order to align block sequences. Lines within a block are matched using the Python library `munkres`, which implements Munkres' improvement on the standard Hungarian matching algorithm to be (strongly) polynomial [25, 26, 33]. We also perform one major optimization over the algorithm. Given the collection of segments to template, SelfStarter first removes all duplicates from the collection — segments whose representation in Batfish's format are identical to that of some existing segment in the collection. Since there are often subsets of the configurations in a role that are meant to be exactly identical, this optimization can significantly improve the efficiency of structured generalization by reducing the number of calls to the expensive matching algorithms.

The order in which segments are considered for templating can affect the final metatemplate and groups produced. Our implementation uses a simple heuristic. The segments are partitioned based on their line counts. The segments in the largest partition (i.e., the partition containing the most segments) are templated first, choosing segments randomly from that partition until it is empty. The remaining segment partitions are then processed in order of decreasing partition size. Intuitively this heuristic tries to template segments that are likely to be outliers last, so that their impact on the templates of other segments is minimized.

The structured generalization algorithm produces a metatemplate in Batfish's vendor-agnostic format, but SelfStarter must output the metatemplate in some vendor-specific format in order to be understandable to network engineers. Currently SelfStarter outputs metatemplates in Cisco's IOS format. This has been sufficient to get feedback for our experiments (§6), since the majority of nodes use that format, and it is similar enough to the other formats for the engineers to understand the results. It would be straightforward to add pretty printers to output the metatemplate in other vendor-specific formats in the future.

Given the results of the structured generalization algorithm, SelfStarter finally produces the output visualization as shown in Figure 8. First the metatemplate lines are partitioned, where two lines are placed in the same subset if and only if for every segment, either both lines are in the segment or neither is. Then a color is chosen for each subset, each inferred template is mapped to the set of colors of its lines, and the colored tables are created and output as HTML files.

[7] `https://github.com/batfish/pybatfish`

6 Evaluation

To evaluate SelfStarter, we applied it to a collection of datacenter networks from a large cloud provider, a wide area network from the same cloud provider, and the campus network of a large university. These networks differ widely in their structure, scale, and management style. Yet SelfStarter identified misconfigurations in all of them; in two of the three networks SelfStarter identified previously unknown errors. Further, SelfStarter's inferred templates closely match any manually written templates that exist for these networks. A dominant cause of the errors that SelfStarter identified were planned configuration updates not yet applied or inconsistently applied. We ran SelfStarter on a 3.6 GHz quad-core machine with 32 GB of RAM.

Methodology: For each network, we obtained a snapshot of the router configurations and partitioned them into roles based on router names, sometimes with the help of the network operator. We then parsed the configurations with Batfish and ran SelfStarter on the output once per *triple*, where a triple is a tuple of a segment type (e.g., ACL), role (e.g., the border routers), and segment name or regular expression (e.g., `border-out-1`). SelfStarter produces a metatemplate for each triple.

Network	Number of routers	Number of roles	Number of triples	Batfish parsing time	SelfStarter running time
Datacenter	$O(10000)$	$O(1000)$	25475	150 min	90 min
WAN	$O(100)$	$O(100)$	21011	14 min	20 min
Campus	106	1	6	2 min	<2 min

Table 1: Networks' parsing and running times

Table 1 shows the order of magnitude of each network in number of routers and roles, along with the number of triples on which we ran SelfStarter[8] and the time for Batfish parsing as well as the total time to run on all triples.

We define a **consistent triple** to be one whose metatemplate consists of only a single group (i.e., all segments meet the same template). Similarly, an **inconsistent triple** is one for which SelfStarter generates a metatemplate with at least two groups. We consider *all* inconsistent triples to be group outliers, and we report the number of such outliers as well as the number that are true/false positives, by comparing with the ground truth (either a "golden" configuration or the

[8] For confidentiality, we cannot disclose the exact number of routers and roles for the cloud provider.

network operator). Perhaps surprisingly, this very coarse way of identifying group outliers is quite effective in identifying real errors in practice, as we show below.

We tried to automatically identify parameter outliers, where a parameter value is considered an outlier if the value's frequency is below a threshold. We considered different threshold functions – $X\%$ of max frequency, $X\%$ of average frequency, etc. — and used an existing technique [28] to identify a good threshold value by finding the point at which the number of outliers spikes as X is varied. However, in the cloud provider networks nearly all parameter outliers that we investigated using this approach were false positives. The global nature of the cloud provider networks makes it likely that there will be many different parameter values, including some used infrequently. Therefore, our experiments do not identify parameter outliers for the cloud provider networks. For the university network, the number of metatemplates and parameters was small enough for us to manually identify parameter outliers and then validate them with the network operators.

6.1 Datacenter Networks

We applied SELFSTARTER to a large collection of production datacenter networks, totaling tens of thousands of devices and hundreds of millions of lines of configuration, from a cloud provider. The datacenters are set up in a folded Clos topology and run the eBGP routing protocol with private AS numbers, as described in an RFC [27]. We obtained a snapshot of device configurations from December 2018.

Ground truth: The configuration files for all network devices are generated automatically from a set of hand-written templates, which are kept up-to-date as "golden" templates. The templates are parameterized, and a separate database maintains the parameter values for each node (e.g., its list of SNMP monitoring servers). Both the golden templates and the parameters database are subject to periodic updates. A software service automatically generates new per-device "golden" configuration based on these updates and installs them on the running devices. However, there are constraints on when and how often it is appropriate to update a device. For example, a device that is a single point of failure must be updated only after customers that could be impacted are safely transitioned. The software service takes these constraints into account when updating the configurations.

We consider an inconsistent triple to be a true positive if the configurations of at least one group of nodes that SELFSTARTER identifies differ from their golden configurations. Because of the software service that automates configuration updates, we did not expect to (and indeed did not) find new errors with SELFSTARTER. Rather, this experiment allows us to objectively validate SELFSTARTER by comparing its results with a well-maintained source of ground truth.[9]

[9]However, in principle SELFSTARTER can still be useful for this network, to catch errors in the automation service itself.

Triples: Recall that a triple contains a segment type, role, and segment name. Since each datacenter network defines ACLs, prefix-lists, and route-policies, we include all three segment types in our triples. Each node's name includes both the name of the datacenter to which it belongs and its *tier* within the datacenter (e.g., top-of-rack); we treat each (datacenter, tier) pair as a unique role. Finally, for each segment type and role, we create a triple for each unique segment name of that type in the role, resulting in more than 20,000 triples.

Segment Type	Consistent Triples	Inconsistent Triples		
		Reported	Investigated	True positives
ACLs	9700	938	400	400
Prefix Lists	2954	0	-	-
Route Policies	11653	230	230	230

Table 2: Datacenter Results

Results: SELFSTARTER identified 1168 group outliers across the three segment types (Table 2), which is fewer than 5% of all triples. We investigated all 230 of the route-policy outliers. We randomly selected 400 ACL outliers to investigate while ensuring that this subset contains at least one triple for each different segment name that appears in the set of outliers. All 630 outliers were determined to be true positives. Specifically, in all cases, at least one of the groups of routers was correctly following the golden template and the difference between configurations was due to a configuration update had been applied to some, but not all, of the nodes in the same role. As mentioned earlier, updates are delayed for many reasons, so such differences are expected and are eventually resolved by the software service.

6.2 Wide Area Network

The WAN we analyzed is one of the largest backbone networks in the world; it interconnects North America, South America, Asia, Europe, Africa, and Oceania. The backbone consists of hundreds of thousands of kilometers of fiber, hundreds of routers, and millions of lines of router configuration. The configurations for the routers are stored in a centralized database, from which we obtained a June 2019 snapshot. All routers are JunOS-based and use the flat-juniper configuration language format.

Ground truth: The WAN does not employ explicit configuration templates. However, the network operators rely partly on scripts to manage the network; the templates are implicit in these scripts. Typically, each script performs one specific task. As an example, a script configures a set of route policies and verifies that they are consistent across a set of devices. In effect, the equivalent of "golden" configurations described for datacenters do not exist for the wide area. Thus, we asked the network operators to help classify SELFSTARTER's outliers as true/false positives.

Triples: We determined useful network roles using an iterative process. We initially divided nodes into roles based on only their network functionality from their names — edge routers, border routers, core routers, route reflector routers and so on. As in the datacenter, we then created one triple for each unique segment name, per segment type and role.

The initial results using this role scheme contained numerous false positives. Upon consulting a network operator, we immediately (within minutes) received the feedback that these roles were too coarse-grained as well as guidance on how to further refine the roles. The first refinement was to take into account the operating environment of each node. For example, two nodes, where one has external peering enabled and the other does not, should not be considered to be in the same role. Fortunately, we were able to utilize information in a file, maintained by the operator, that lists the operating environment of each node. The second refinement was to take into account geographic location. Routers have certain specializations based on geographical regions in order to meet local policies such as government-specific privacy requirements.

In the end, the number of routers is only roughly $5\times$ the number of roles. In retrospect this makes sense for the WAN due to its global spread. Devices necessarily have many policy differences across regions, for instance based on local peering relationships. Though our initial guess of roles was overly coarse, it was trivial for the operator to immediately identify the issue and provide refined role information.

The WAN defines ACLs, prefix lists, and route policies. However, to date we only have feedback from the operators about SELFSTARTER's output for prefix lists and route policies; thus we omit ACLs from the results below.

Segment Type	Consistent Triples	Inconsistent Triples		
		Reported	Investigated	True positives
Prefix Lists	10042	166	138	7
Route Policies	10969	56	33	33

Table 3: WAN Results

Results: SELFSTARTER identified 222 inconsistent triples (Table 3), which is $\sim$1% of all triples. All outliers that were flagged as true positives were previously unknown and have since been remediated by the operators. All 33 investigated triples for route policies were verified as true positives by the network operators. There were several different root causes. Interestingly, one class of errors was due to bugs in the automation scripts. Specifically, a script that checks for the existence of certain commands in a route policy accidentally did not consider the order of these commands. Since SELFSTARTER takes order into account, its metatemplate contained multiple groups and hence identified ordering errors that change the behavior and that the script missed. Another class of errors, which caused five inconsistent triples, was due to a spurious community tag being added to some route announcements in a few routers.

SELFSTARTER was much less effective at finding real errors in prefix lists, with only 7 out of 138 investigated triples determined to be true positives. In 95 out of 138 triples, the Juniper command `apply-path` is used to create a prefix list by expanding an existing set of addresses defined elsewhere in the configuration, such as the set of local IPs or NTP servers. The seven true positives represented cases where there were inconsistent sets of loopback interfaces defined on different routers, and required cleanup. For the other 88 cases, the operator informed us that it is expected that every router's prefix list will expand to a different set of IPs, and that they can have different numbers of such addresses. For the remaining prefix-lists, the operator informed us that those prefix-lists are locally significant on every router and will always be different. Since SELFSTARTER creates multiple groups whenever two segments have different numbers of lines, this led it to report these spurious inconsistent triples.

On the positive side, it was easy for the operator to quickly understand SELFSTARTER's output and identify these cases as false positives. In fact, the total operator time to classify all of SELFSTARTER's results, for both prefix lists and route policies, was under 30 minutes. Going forward, it would be simple to allow operators to suppress errors reported for segments/roles with well-known differences.

6.3 University Network

The university network consists of approximately 1000 devices, including border routers that connect to external ISPs, core routers that form the backbone, and building routers that handle connectivity for individual buildings. We obtained a snapshot from May 2019 of the network configurations.

Ground truth: Configurations in the network were created using a mix of manual setup and templates. However, even where templates exist, the configurations are still updated directly over time to meet evolving policy needs, and the original templates are not always kept up to date. Therefore, we asked the network engineers to validate our results.

Triples: Node names include a two-letter abbreviation to indicate the network role, for example *cr* for core routers and *br* for building routers. We analyzed only the 106 building routers in the network. This network role was chosen because it is the most interesting one for our purposes – it is the only role that contains many segments that are intended to be similar to one another. Specifically, the role contains six distinct ACLs that appear in nearly all routers. ACLs account for more than one third of the lines in these configurations. The building routers do not contain prefix lists or route policies.

We ran SELFSTARTER with a regular expression for each ACL rather than an exact name, since they have slightly different names in each router. Specifically, the name of each router's version of an ACL includes the router's associated building name and the date on which the ACL was created.

ACL Pattern	Number of ACLs in largest group	Number of ACLs in other groups
br_aux_mgmt_*_in_*	88	18
br_mgmt_*_in_*	82	24
br_wlan_mgmt_*_in_*	80	24
br_aux_mgmt_*_out_*	84	22
voip_*_in_*	61	22
voip_*_out_*	61	22

Table 4: May 2019 "br" router ACL results.

Results: Table 4 summarizes the results. For each ACL SelfStarter identified one dominant group of nodes that share a common template, along with one or more smaller groups of nodes that have a different template. Hence non-dominant groups potentially indicate misconfigurations.

To date the network engineers have provided feedback on the first three ACLs in the table. In all three cases the network engineers have confirmed that the group outliers that SelfStarter identified are indeed misconfigurations: every ACL that SelfStarter placed in a non-majority group has at least one misconfiguration.

For example, the metatemplate and the groups that SelfStarter inferred for the first ACL regex in Table 4 was shown in Figure 1. The 18 ACLs in Groups 2 and 3 include some old `deny` lines. Initially, IP spaces 14.10.49.0/24, 14.10.50.0/24, and 15.8.228.0/20 were allocated for infrastructure management; thus access to them was restricted using `deny` lines (ACL Lines 3 – 5). However, that allocation changed at some point to 14.10.0.0/19 and 17.7.240.0/20; thus the intent was to deny traffic to these new IP spaces (ACL Lines 1 – 2) and remove the old `deny` lines. Because it is each department's responsibility to update the ACLs for their building routers, some of the ACLs were not properly updated. The second and third ACLs in Table 4 exhibited similar misconfigurations, all of which were confirmed by the network engineers.

The network engineers have templates for these ACLs and so we manually compared the template that SelfStarter inferred for the dominant group with those templates for the first three ACLs in Table 4. For the first two ACLs, SelfStarter's template matches the network's hand-written templates. Specifically, the templates are *line-for-line identical*, except that in some cases SelfStarter's template has a concrete value where the golden template has a parameter. For example, SelfStarter might learn that a particular line uses IP address `1.2.3.A`, since all segments agree on the first three octet values, but the hand-written template treats the entire IP address as a parameter.

For the third ACL, SelfStarter's template for the dominant group does not match the network's template. Specifically, the network's template has two additional `permit` lines (one based on IP and another on ICMP). The network engineers informed us that this template was indeed stale. This shows that a possible use case for SelfStarter is to automatically identify "template drift".

Finally, we manually identified three parameter outliers in the first ACL. Earlier we said that parameter `A` in line 6 (ACL Line 6) of the metatemplate in Figure 1(a) was confirmed as a misconfiguration. We also asked the network engineers about parameters `B` and `E` (line 7). Both of these turned out to be intentional differences. For example, in the case of parameter `E` two building routers required more hosts than 255 and so were allocated a larger IP space than the other routers.

7 Discussion

Our experience applying SelfStarter to real-world networks and interacting with the operators has led to several observations, which we discuss here.

SelfStarter assumes that many router configurations in a network are *structurally similar* to one another. Our experiments largely bear out this assumption, since network operators typically employ configuration templates or common guidelines to simplify the configuration of groups of nodes. However, when this assumption does not hold, for example in the prefix lists of the WAN that we analyzed, then SelfStarter will not be as useful.

Our experimental evaluation indicates that SelfStarter can be useful in different kinds of networks, which are managed in different ways. Even where templates exist, they can become stale over time as the running configurations are updated. Even where automation exists, the automation can be incomplete or itself be a source of errors. Because SelfStarter takes as input only the final per-node configurations, it provides a useful form of redundancy for validating configurations, regardless of how they were created.

Finally, SelfStarter's structural approach to outlier detection means that it cannot determine the *behavioral* differences between outliers and non-outliers. However, in our experience a key advantage of SelfStarter is that its results are very easy for operators to understand, precisely because a metatemplate retains the structure of the original configuration segments. All of the network operators with whom we interacted were able to quickly decide whether an identified outlier was a true or false positive.

8 Related Work

To our knowledge, ours is the first technique to automatically infer templates for network configuration segments. We use these templates to identify misconfigurations, so we compare against other techniques for doing so.

Network Verification Network verification for the data plane (e.g., [6,11,23,24,29,31,39]) and control plane (e.g., [7, 14–16,18]) models the *semantics* of the network, which allows them to verify deep behavioral properties. However, they rely on the user providing a formal specification, and otherwise are limited to checking generic properties like the absence of loops. In contrast, SELFSTARTER leverages the *structural* similarity of many router configurations to identify network-specific errors without a specification, but SELFSTARTER cannot map these errors to specific undesired behaviors.

Minesweeper [7] can verify *functional equivalence* of two router configurations, which could be used to identify outliers. However, exact functional equivalence is much too strong a criterion in general and so would lead to a high false positive rate and make it harder to spot errors. For example, partitioning the 88 ACLs of Group 1 from Figure 1, all of which are correct, by functional equivalence would result in 44 groups, each of size 2. On the plus side, by modeling a segment's behavior Minesweeper can safely allow reorderings that SELFSTARTER would spuriously flag, for example swapping the order of unrelated `permit` and `deny` lines.

Outlier Detection for Network Configurations: El-Arini and Killourhy [12] use a form of Bayesian inference to identify outlier configuration lines and demonstrate that the approach can find "lone commands," lines that only appear once in the given set of configurations. Le *et al.* [28] employ a data-mining algorithm to infer association rules; configurations that violate the rules are deemed outliers. These approaches identify misconfigurations in settings of interface definition and BGP sessions, including BGP route policies.

However, these approaches share two key limitations. First, they can only find outliers based on *exact equivalence*. Specifically, neither the approach of El-Arini and Killourhy nor the inferred association rules of Le *et al.* can infer configuration *parameters*, which is necessary to account for values that differ in expected ways across devices. Second, neither approach takes into account the importance of line reorderings within a configuration segment. Our structured generalization algorithm overcomes both limitations: bipartite matching of lines induces a natural form of parameterization, and sequence alignment of blocks enforces ordering constraints.

Mining Configuration Policies: Benson *et al.* show how to infer reachability policies for a network data plane [9]. Recently, Birkner *et al.* show how to infer similar policies that also take into account the control plane, ensuring that policies hold even in the presence of failures [10]. These *semantic* policies, which pertain to the end-to-end behavior of the network, are orthogonal and complementary to SELFSTARTER's *structural* policies for network configurations.

Benson *et al.* also introduce metrics and techniques to gauge the *complexity* of configurations [8]. Closest to our work is their technique to infer network *roles*. Their algorithm replaces field values with dummy entries — for example, IP addresses with the string "IPADDRESS" — and then employs an off-the-shelf clone detection tool [22] to find similar configurations. Our structured generalization is similar in spirit but provides several refinements necessary for template inference, including fine-grained support for parameterization and reordering. However, their work is complementary to ours, as we require the user to provide network role information.

Diagnosing Misconfigurations: Another line of work focuses on diagnosing the cause of misconfigurations. For example, NetPrints [5] does this through a form of decision tree learning, and PeerPressure [38] does this through a statistical analysis. Unlike SELFSTARTER, these tools start from a set of known or suspected misconfigurations, which the user must supply. Further, these tools diagnose misconfigurations in terms of a set of simple configuration features, while SELFSTARTER leverages the full structure of the configuration segments through template inference.

Automatic Differencing: Many algorithms exist for "diffing," for example, text comparison [32], clone detection [21, 22], and sequence alignment [36]. Our *structured generalization* algorithm combines some of these techniques in a novel manner, based on our domain requirements. We employ bipartite matching at the line level to support permutation and parameterization, but we introduce the *block* abstraction and perform sequence alignment on blocks to restrict certain reorderings while admitting insertions and deletions.

9 Conclusion

We presented an approach to identify misconfigurations in complex configuration segments, such as ACLs and route policies, without a specification. Such segments are typically intended to be similar across nodes playing the same *role*, yet they often have many intentional differences. We address this challenge by automatically inferring *templates*, modeling the (likely) intentional differences as variations within a template and the (likely) erroneous differences as variations across templates. Our *structured generalization* algorithm employs a novel two-level matching technique to allow controlled forms of parameterization and reordering within templates. To our knowledge this is the first approach to automatic template inference for network configuration segments.

Unlike the majority of work in network verification, which reasons about the *semantics* of networks, SELFSTARTER's analysis instead reasons about the *structure* of their configurations. While SELFSTARTER by design cannot understand packet behavior, it makes up for this lack by providing concise, actionable feedback directly in terms of the configurations. As a result, it has helped operators find and fix previously unknown network misconfigurations in three very different types of networks: datacenter, WAN, and campus.

Acknowledgments

Thanks to the NSDI reviewers and our shepherd Laurent Vanbever for their insightful comments, to the anonymous network operators for their feedback on SELFSTARTER's results, and to Shuang Zhang for help with SELFSTARTER's output visualization. This work was partially supported by NSF grants CNS-1704336 and CNS-1901510.

References

[1] Cisco blog | bgpmon.
https://bgpmon.net/blog/.

[2] Hyperscale cloud reliability and the art of organic collaboration.
https://www.microsoft.com/en-us/research/blog/hyperscale-cloud-reliability-and-the-art-of-organic-collaboration/.

[3] Intent-based networking in the data center: Cisco vs. juniper.
https://www.datacenterknowledge.com/networks/intent-based-networking-data-center-cisco-vs-juniper.

[4] Intent-based networking startups.
https://www.datacenterknowledge.com/networks/4-intent-based-networking-startups-innovating-disrupt-data-center-network.

[5] Bhavish Agarwal, Ranjita Bhagwan, Tathagata Das, Siddharth Eswaran, Venkata N. Padmanabhan, and Geoffrey M. Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 349–364, 2009.

[6] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, SafeConfig '10, pages 37–44, New York, NY, USA, 2010. ACM.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 155–168, New York, NY, USA, 2017. ACM.

[8] Theophilus Benson, Aditya Akella, and David Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association.

[9] Theophilus Benson, Aditya Akella, and David A. Maltz. Mining policies from enterprise network configuration. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 136–142, New York, NY, USA, 2009. ACM.

[10] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Config2spec: Mining network specifications from network configurations. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 2020.

[11] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. ddnf: An efficient data structure for header spaces. In Roderick Bloem and Eli Arbel, editors, *Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*, volume 10028 of *Lecture Notes in Computer Science*, pages 49–64, 2016.

[12] Khalid El-arini. Bayesian detection of router configuration anomalies. In *In Sigcomm Workshop on Mining Network Data*, pages 221–222. ACM, 2005.

[13] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[14] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 217–232, Savannah, GA, 2016. USENIX Association.

[15] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 43–56, Berkeley, CA, USA, 2005. USENIX Association.

[16] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, Oakland, CA, 2015. USENIX Association.

[17] Fortune. American airlines network outage delays flights nationwide.
http://fortune.com/2018/07/29/american-airlines-network-outage/.

[18] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the*

2016 ACM SIGCOMM Conference, SIGCOMM '16, pages 300–313, New York, NY, USA, 2016. ACM.

[19] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 350–361, New York, NY, USA, 2011. ACM.

[20] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 200–213, New York, NY, USA, 2019. ACM.

[21] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[23] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[24] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.*, 42(4):467–472, September 2012.

[25] H. W. Kuhn. Variants of the hungarian method for assignment problems. *Naval Research Logistics Quarterly*, 3(4):253–258, 1956.

[26] H. W. Kuhn and Bryn Yaw. The hungarian method for the assignment problem. *Naval Res. Logist. Quart*, pages 83–97, 1955.

[27] P. Lapukhov, A. Premji, and J. Mitchell. Use of BGP for routing in large-scale data centers. Internet draft, 2015.

[28] Franck Le, Sihyung Lee, Tina Wong, Hyong Kim, and Darrell Newcomb. Detecting network-wide and router-specific misconfigurations through data mining. *Networking, IEEE/ACM Transactions on*, 17:66 – 79, 03 2009.

[29] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 499–512, Oakland, CA, 2015. USENIX Association.

[30] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, pages 3–16, New York, NY, USA, 2002. ACM.

[31] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[32] Meld. Compare files, directories and working copies. `https://git.gnome.org/browse/meld/tag/?h=release-0_5_1`.

[33] James Munkres. Algorithms for the assignment and transportation problems, 1957.

[34] GD Plotkin. A note on inductive generalization, machine intelligence , editors b. *Meltzer, DI lichine, University PresB, Edinburgh*, Vol. 5:153–163, 1970.

[35] The Register. How four rotten packets broke centurylink's network for 37 hours, knackering 911 calls, voip, broadband. `https://www.theregister.co.uk/2019/08/20/centurylink_outage_report_fcc/`.

[36] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[37] Ars Technica. Google goes down after major bgp mishap routes traffic through china. `https://arstechnica.com/information-technology/2018/11/major-bgp-mishap-takes-down-google-as-traffic-improperly-travels-to-china/`.

[38] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with peerpressure. In *6th Symposium on*

Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004, pages 245–258, 2004.

[39] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Trans. Netw.*, 24(2):887–900, April 2016.

`tpprof`: A Network Traffic Pattern Profiler

Nofel Yaseen
University of Pennsylvania
`nyaseen@seas.upenn.edu`

John Sonchack
University of Pennsylvania
`jsonch@seas.upenn.edu`

Vincent Liu
University of Pennsylvania
`liuv@seas.upenn.edu`

Abstract

When designing, understanding, or optimizing a computer network, it is often useful to identify and rank common patterns in its usage over time. Often referred to as a network traffic pattern, identifying the patterns in which the network spends most of its time can help ease network operators' tasks considerably. Despite this, extracting traffic patterns from a network is, unfortunately, a difficult and highly manual process.

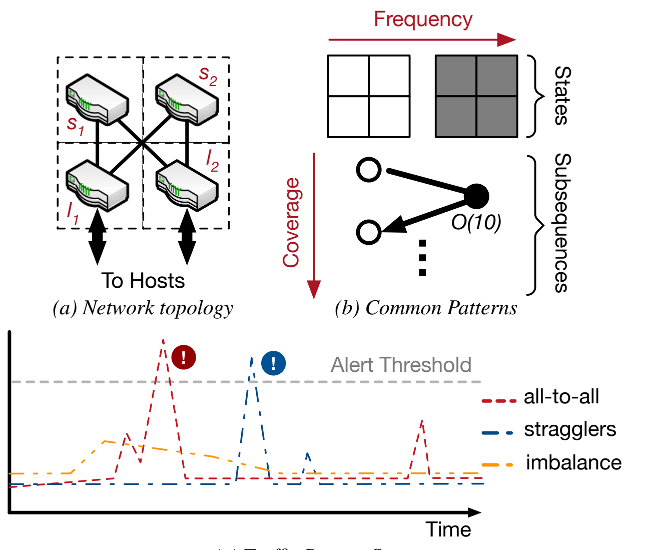
In this paper, we introduce `tpprof`, a profiler for network traffic patterns. `tpprof` is built around two novel abstractions: (1) network states, which capture an approximate snapshot of network link utilization and (2) traffic pattern subsequences,which represent a finite-state automaton over a sequence of network states. Around these abstractions, we introduce novel techniques to extract these abstractions, a robust tool to analyze them, and a system for alerting operators of their presence in a running network.

1 Introduction

When designing, understanding, or optimizing a computer network, it is often useful to identify common patterns in its usage over time. Often referred to as a **network traffic pattern**, identifying the patterns in which the network spends most of its time can result in useful insights:

- *All-to-all traffic*, which might manifest as uniform utilization of all paths between a set of application nodes, might suggest the importance of bisection bandwidth and guide future provisioning decisions.
- *Chronic stragglers*, where we expect all-to-all traffic but a significant amount of time is spent with only a few flows active, might suggest the need for better sharding and mitigation techniques.
- *Elephant flow dominance*, in which utilization is dominated by isolated path-level hotspots, might guide future provisioning decisions.
- Finally, *synchronized requests/responses*, indicated by repeated bursts of cross-network communication all originating at a single node, might motivate changes in the application or network architecture.

While there are a number of existing tools that capture flow- and switch-level trends (e.g., heavy hitter analysis [68], network tomography [28], or the vast array of network analytics suites on the market [1–6, 30]), identifying prevalent *network*-level patterns typically requires a significant amount of manual effort and specialized analyses. To determine the presence of synchronized requests/responses, for instance, an operator might need to instrument the start and stop times of all flows in the system, correct for the time drift of different machines, compute the cluster tendencies of the data (e.g., with a Hopkins statistic or heuristic), and distinguish it from all-to-all traffic by examining the sources and destinations of synchronized flows. To determine whether this pattern is a particularly common one would require additional analyses.



(c) Traffic Pattern Scores

Figure 1: `tpprof`'s visualizations for b common traffic patterns and c the TPS score over time for a simple leaf-spine topology, a. We describe these in more detail later, but in b, states are heatmaps of common utilization patterns over the network in a; darker is hotter. Subsequences are common transition patterns between the aforementioned states. These are ranked by their frequency of occurrence and their cumulative coverage of the profiled run, respectively. The subsequence shows an all-to-all pattern: the network starts unutilized (left state), becomes fully utilized (right state) for 10s of samples, then returns. In c, `tpprof` is tracking three different known traffic patterns. When the score of any of them crosses the alerting threshold (twice in the figure), `tpprof` deduces that the pattern has occurred in the network.

Our goal in this work is a tool for the automatic identification of the most prevalent traffic patterns in a network. To that end, we present the design and implementation of `tpprof`, a network traffic pattern profiler.

Similar to traditional application profilers like gprof [27] or Oprofile [22] that have helped programmers understand and improve their software for decades, `tpprof` automatically measures, extracts, and ranks common traffic patterns of individual applications within running networks. It also facilitates the monitoring of known patterns so that, when specific patterns appear in the network, the operator is informed. It does both of these things without modifying applications and without affecting existing network traffic—the only changes we require are to switch monitoring configurations. Examples of both of the above tools in action are shown in Figure 1.

Traffic patterns are, unfortunately, significantly more challenging to profile than applications. Traditional profilers benefit from well-defined building blocks (functions or lines of code) connected by well-defined call graphs. In contrast, networks offer little such structure: switch and link utilizations are noisy and measured in real values (Bps); their evolution over time is even less constrained. In the end, two different instances of something as simple as all-to-all traffic will never look exactly the same.

Thus, `tpprof` is built around two novel abstractions: (1) *network states*, which capture an approximate snapshot of a network's device-level utilization and (2) *traffic pattern subsequences*, which represent a finite-state automaton over a sequence of network states. As hinted above, subsequences serve as both output and input to our system: output in the case of profiling an existing network, input in the case of specifying a traffic pattern alert. In both cases, classification of network states and sub-sequences is approximate and implemented through specialized clustering techniques.

We implement and deploy `tpprof` to a small hardware testbed in order to monitor and profile the traffic patterns of real distributed applications like memcache, Hadoop, Spark, Giraph, and TensorFlow. We demonstrate that, using `tpprof`, we can find meaningful patterns and issues in their behavior. Further, we demonstrate `tpprof`'s utility on larger and more complex networks by profiling a trace taken from one of Facebook's frontend clusters. While our evaluation focuses on data center networks (where interesting and impactful distributed applications are plentiful), `tpprof` and its techniques generalize to arbitrary networks.

Specifically, this paper makes the following contributions:

- **Novel abstractions for describing common traffic patterns:** We introduce two abstractions, network states and traffic pattern subsequences, that together enable network operators to easily describe and reason about common traffic patterns. Network states capture similar configurations of approximate utilization of a specific application or set of applications running in a network. Subsequences are then strings of states with bounded repetition that summarize traffic pattern changes over time.
- **Domain-specific algorithms for clustering and ranking both network states and subsequences:** Through empirical analysis of a variety of application traffic patterns, we identify and design algorithms that transform a network trace into the building blocks of traffic patterns. Specifically, we demonstrate through PCA and waypoint analysis of real application traffic that GMMs are well-suited to capturing first-order similarities between different network utilization patterns. In the case of subsequences, we create a domain-specific clustering algorithm that extracts sequences that are both common and that provide broad coverage of the measured network traces.
- **A language and mechanism for expressing and fuzzily matching known traffic patterns in observed traces:** Finally, to complement the above, we develop a simple grammar for describing traffic patterns and introduce an algorithm that automatically identifies approximate occurrences of known traffic patterns within network traces. Our scoring engine outputs a confidence score that can be used to generate alerts when known traffic patterns appear in observed traces.

Taken together, `tpprof` is, to the best of out knowledge, the first profiling tool for network-wide traffic patterns. Our implementation is in Python and the code is open source.[1]

2 The Anatomy of a Traffic Pattern

We begin by introducing the definitions and abstractions on which `tpprof` is built. First and foremost, we define the overall traffic pattern of a network as follows:

NETWORK TRAFFIC PATTERN — A function $f(x,t)$ that represents, for an entire network N across a time span $[t_0, t_1]$, the utilization of device $x \in N$ at time $t_0 \leq t \leq t_1$.

We also define, for each application in the network:

APPLICATION-SPECIFIC TRAFFIC PATTERN — $f_A(x,t)$, equivalent to the network traffic pattern, but only accounting for a single application or set of applications, A.

For the purposes of our clustering and ranking algorithms, the distinction is unimportant; unless otherwise specified, we use 'traffic pattern' to refer to both. Instead, the choice of whether to filter by application is entirely the user's (with the mechanisms in Section 4); Regardless, for a given network and overall workload, we note that the traffic pattern of both the network and its constituent applications will typically exhibit predictable and repeated characteristics given a sufficiently long measurement period. These patterns can occur over short time spans of individual packets and flows, or over longer time spans in the form of diurnal or weekday/weekend effects.

A contribution of this paper is the decomposition of traffic patterns into a more convenient low-level primitive:

[1] https://github.com/eniac/tpprof.

Network Sample — $|N|$ real values that capture an approximate snapshot of $f(x,t)$ for all devices $x \in N$, at a particular time t, and averaged over the last t_Δ seconds.

Network Sample Sequence — A chain of network samples that sample $f(x,t)$ over increments of t_Δ, where t_Δ is bounded by the measurement granularity of the system.

Any traffic pattern can be described in these terms. For the network in Figure 1a, the all-to-all pattern in Figure 1b is one example. Another is chronic stragglers, which we can describe as a transition between two configurations, assuming a load balanced network: (1) all switches at high utilization and (2) only l_1, l_2 and s_1 at high utilization; or the same but replacing s_1 with s_2.

We can perform a similar exercise for all of the many (possibly application-specific) traffic patterns in the literature, e.g., rack-level hotspots in data centers [24, 29, 44, 58], synchronized behavior of distributed applications [9, 20, 67], and stragglers in data-intensive applications [21, 41, 43]. We do the same for the link- and switch-level traffic patterns that are the focus of most existing automated profiling tools [1–6, 30].

While not necessarily the way these patterns were described originally, sequences of network samples provide a general primitive with which we can represent arbitrary patterns.

3 `tpprof` Design Overview

Our goal in this paper is the design and implementation of a *profiler* for network and application-specific traffic patterns. Our system, `tpprof`, is intended to identify traffic patterns, rank them in prevalence, and assist network operators in monitoring for their recurrence. Like other profiling tools, `tpprof` is not intended to improve networks directly; rather, its focus is on assisting users with designing, understanding, and optimizing them.

On that note, we take inspiration from traditional sampling profilers like gprof [27], Oprofile [22], and Valgrind [48]. These profilers take an unmodified application and they periodically sample system state (e.g., stack traces) to produce a statistical profile of the target application. Early instantiations solely sampled program counters; over time, they expanded to capture trends in function utilization and call graph traversal.

`tpprof` uses a similar approach to construct profiles of traffic patterns. To that end, network samples and sequences of samples present an attractive substrate. In principle, a sequence of network samples create a statistical profile of an application's network utilization. Unfortunately, these samples are unlikely to ever repeat: small differences in application processing time, workload, and background traffic can cause substantive differences in traffic, as can slight noise in the sampling frequency of the measurement framework. Extracting patterns from raw samples is challenging.

Core abstractions. To address this challenge, we introduce two additional abstractions:

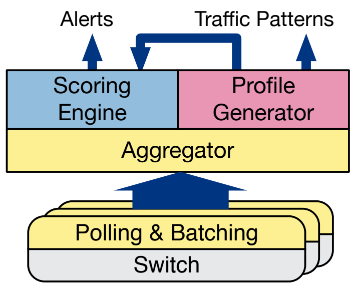


Figure 2: The overall architecture of `tpprof`. `tpprof` polls, batches, and aggregates switch counters from the network. These are fed into (1) a scoring engine that alerts on detection of known patterns and (2) a profile generator that extracts common traffic patterns from the gathered trace.

Network State — A class of network samples defined by a single, n-device network sample, S, and n variance values, $\bar{v}$ such that S is a centroid of the multivariate normal distribution with shape defined by $\bar{v}$.

Network State Subsequences — A class of sequences of network states that allows for bounded repetition of states. A state subsequence can be represented as a regular expression or finite state automata of network states.

Multiple network samples can be mapped to a single network state and multiple sample sequences can be mapped to a single state subsequence. These abstractions are tolerant to noise by design: variations of link utilization from sample to sample are smoothed by our method of extracting network states; variations in the evolution of those samples over time are smoothed by our method of extracting subsequences. The precise construction of both of the above elements is described in Sections 5.1 and 5.2.

Components. `tpprof` consists of three primary components:

1. A configurable *sampling framework* that periodically samples the device-level utilization of a specified application, set of applications, or the full network (Section 4).
2. A *profiling tool* for the automatic extraction and visualization of the most common states and state subsequences in the captured data (Section 5).
3. An *alerting system* that scores incoming traces against a set of user-defined patterns using a fuzzy string search in order to facilitate network automation (Section 6).

Of the above, only (1) affects the network itself; (2) and (3) occur out-of-band. As such, the overhead of `tpprof` is minimal: in the case of an non-application-specific traffic pattern, little is required beyond an SNMP poller; application-specific patterns only require simple `iptables` and switch configuration changes on top of that.

Our current implementation leverages programmable switches and a recently proposed network-wide monitoring framework [63]. This provides slightly more control and accuracy than an implementation based on top of traditional switches, but it is not a strict requirement; we detail both approaches in Section 4.

Workflow. `tpprof` profiles production networks. A typical workflow thus proceeds as follows. First, users specify three configuration parameters: the start time a, the end time b, and the sampling interval i. The network can optionally be configured to track certain applications separately. Regardless, a centralized service periodically polls the byte counters of the entire network between time a and b, with interval i.

The centralized service will stream the data through a set of scoring algorithms that quantify the prevalence of a set of target patterns in the measured trace. If the score of the trace exceeds a threshold for a given pattern, an alert will be generated. By default, the measurement data is not stored. This changes when users request a profile, i.e., a visualization, of common traffic patterns in the network. In this case, raw network samples are stored for a specified profiling duration for clustering and analysis. The resulting profile can be used to construct additional pattern alerts or analyzed separately. The remainder of this paper describes each of the three components of `tpprof` in more detail.

4 Sampling Framework

`tpprof`'s sampling framework continually polls a custom set of switch counters to capture traffic patterns. Most production networks already implement some form of this—`tpprof` can piggyback on these existing polling suites. `tpprof` is, however, parameterized by at least two configuration options.

- *Application filters:* To profile application-specific traffic patterns, users must provide a proper filter for the traffic in question. In `tpprof`, this takes the form of `iptables` rules. Any filter that can be expressed as an `iptables` rule is allowed. Thus, multiple applications can be captured by a single filter and different flows from the same application can be split into different filters by port, packet type, etc. All traffic matching installed filters are marked with a special set of bits, e.g., in the DSCP field of the packet header. We term the value of these bits a *filterid*.
- *Sampling interval:* Users must also specify an interval, t_Δ, at which `tpprof`'s sampling framework will poll all devices in the network. This interval is common to the entire system, so the network and all application-specific traffic patterns will be read at this rate. Though this is a user-defined value, we anticipate that it should be set to the minimum value feasible for the target network without incurring sample loss. We note that, because the raw data is discarded after alert pattern matching, measurement data storage capacity is not a bottleneck in `tpprof`.

4.1 Counter Implementation and Sampling

Network devices in a `tpprof`-enabled network track a set of device-level application-specific byte counters corresponding to the space of possible *filterid*s. For every packet traversing the switch, the counter associated with the specified *filterid* is incremented by the size of the packet; all categories summed will give the cumulative byte counter of the device. In this design, the network is never reconfigured; instead, users associate applications to *filterid*s directly through the `iptables` rules at every end host.

`tpprof` samples these counters at an interval of t_Δ via a recently proposed measurement primitive, Speedlight. For brevity, we omit the details of its operation and refer interested readers to its non-channel-state variant [62, 63]. At a high level, the primitive is that of a synchronized, causally consistent snapshot of network-wide switch counters. Compared to SNMP and other naïve poling tools, Speedlight provides increased accuracy and low minimum sampling interval, both of which are useful when profiling network traffic patterns.

Alternative implementations. We note that, at its core, the only requirement of `tpprof` is for configurable counters and a method to periodically poll all such counters in the network. There are other implementations that satisfy this requirement.

For instance, most modern switches typically include support for configurable ACL entries with per-entry counters. This approach has the advantage that it can be implemented without end host cooperation. Class of Service (CoS) counters are similarly promising. Note that, if application-specific tracking is not required, periodic SNMP polling is sufficient.

4.2 Batching and Aggregation

While it is possible to directly transmit polled counter results to a centralized profiling service, the scale of measurement data collected by `tpprof` necessitates careful handling. In particular, there are two issues we must address: decreasing overhead and handling sample loss.

For the first, to decrease the number of messages and the overhead per sample, `tpprof` agents running on each network device assemble results locally before shipping batches of size B in the following format:

```
sampleBatch: {
  switch: <SWITCH_ID>,
  indexes: [i : <SAMPLE_ID> for i from 0 to B],
  app1_bytes: [i : <BYTE_COUNT> for i from 0 to B],
  ...
  appM_bytes: [i : <BYTE_COUNT> for i from 0 to B]}
```

`indexes[k]` and `*_bytes[k]` should correspond to a single network sample. Gaps in the samples, e.g., from failures or measurement packet drops, will manifest as gaps in the `indexes` array. In these cases, `tpprof` attempts to interpolate values by taking the difference between byte counters before and after any gap and averaging the difference over the length of the gap. If the device has rebooted or if it stays down for too long, we will treat the device as 'failed' during the missing measurement intervals. 'Failed' devices are excluded from

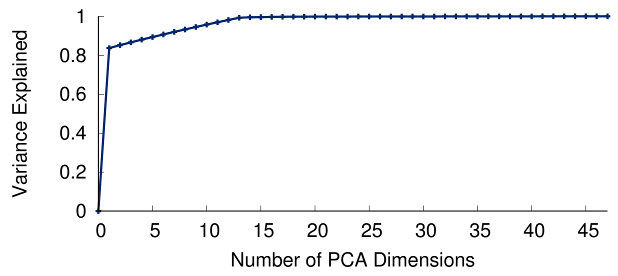


Figure 3: Covariance explained by different numbers of PCA dimensions. Dataset is a trace of utilization over 48 ToR switches in a Facebook frontend cluster.

profiling and treated as wildcards during alerting. Note that reboots are also excluded from interpolation as we do not know how much traffic was sent before the counter was reset.

Storing data for profiling. While `tpprof` does not store raw counter values in the common case, raw values are necessary for generating profiles. Profiles are, therefore, executed on-demand using the API:

```
start_profile(start, end, filter_id)
```

The duration of collection should be long enough to capture a representative slice of behavior. In general, longer is better, but this may be subject to limitations of sample storage space and the user's timeline. `filter_id` $= -1$ indicates the sum of all application-specific counters.

5 The `tpprof` Profiling Tool

We first discuss how `tpprof` extracts and ranks traffic patterns before delving into the scoring and alerting system in Section 6. To that end, the output of the previous subsection (4.2) is a network sample sequence, i.e., a sequence of n-device samples of network utilization. Using that, the output of the `tpprof` profiler is a ranked list of network states and a ranked list of state subsequences, as sketched by Figure 1b and demonstrated in Section 7. `tpprof` achieves this using a pair of domain-specific clustering techniques that are designed to capture first-order patterns in network traffic.

5.1 Network States

The first challenge in identifying meaningful traffic patterns is the inherent noise present in a trace of network samples. Small variations in workload, TCP effects, background traffic, or any number of other factors mean that, most likely, no two network samples will look exactly alike.

To de-noise the data, `tpprof` summarizes network samples into a small number of distinct network states. We can naturally frame this as a clustering problem, where the points to be clustered are the n-element vectors representing network samples. Clustering has been used to great effect in a number of fields, from image segmentation to recommendation systems and anomaly detection; each of these has their own set of challenges and associated clustering algorithms.

Network state extraction is no different in that regard. In this work, we leverage empirical analysis of a variety of applications and traces to identify and design algorithms suited to the domain. Applications observed include Hadoop, Giraph, TensorFlow, Spark, Memcache, and a trace from a production Facebook frontend cluster (see Section 7 for their details).

Dimensionality reduction. Before delving into `tpprof`'s clustering algorithm, we note that, in general, networks present a particular challenge to clustering because of their high device counts. Profiling the ToRs of a 48-rack data center cluster, for instance, might result in a 48-feature input vector, which prior work has indicated might be too many dimensions for typical distance metrics [17].

The general solution to this well known 'curse of dimensionality' [15] is transforming the data into a lower dimensional space before clustering. The simplest approach is to cluster on only a subset of features. While this works in other domains, it is not well suited for our problem because the load on *every* device may be important. Instead, `tpprof` preprocesses data with Principle Component Analysis (PCA) [25], which derives a small set of features that are an orthogonal linear transformation of the original features. Said differently, PCA removes redundancies in the original data by creating a new set of independent features that explain most of the variability in the original data.

PCA is most effective when features are strongly correlated, and there is good reason to believe that this is true in our domain. Recent work [21] shows that network usage is highly correlated, driven by the data-parallelism in distributed systems [31, 65]. Analysis of the Facebook traffic trace verifies this: each ToR had strong and statistically significant correlations ($r > .7$, $p < .001$) with an average of 3.25 other ToRs. The applications we profiled showed similar results.

Figure 3 measures the effect of PCA on that data, gauged by plotting number of PCA dimensions (i.e., features) versus explained variance. All other traces we obtained showed similar results. A value of 1 means that a PCA transformation to K dimensions preserved all the information contained in the original data with 48 dimensions. Even for this large and complex trace, one dimension already explains over 80% of the variance and two dimensions explain ∼85%. Striking a balance between clustering efficacy and explained variance, `tpprof` projects all data into 2D by default. This can be adjusted depending on the data.

Gaussian Mixture Models (GMMs) for sample classification. `tpprof` clusters around typical network variations through its use of GMMs. To demonstrate this effect, we consider the 2D PCA projections described above and visualized, for our set of profiled applications and traces, in Figure 4. To help interpret points in the PCA space, we also plot network load at 4 extreme *waypoints* along the convex hull of each trace. We observe two general cluster shapes in the projected

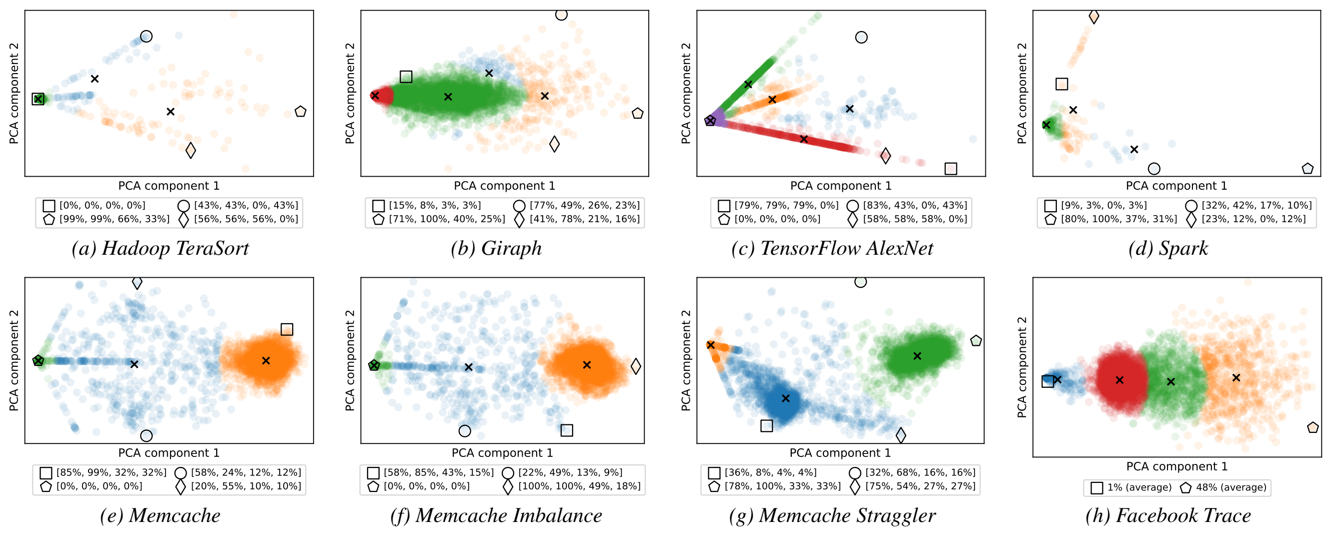


Figure 4: Network samples projected into a 2-dimensional PCA space. Cluster centers are marked with x's. Shaped-markers map points in the space to sample vectors [l1, l2, s1, s2] (see Figure 1a) or, for the Facebook trace, average utilization.

data: 'rays' and 'clouds'.

- **Rays**, like the ones prominent in Figures 4a and 4c, are typically associated with rising or falling utilization on a set of highly correlated nodes. We can see this effect most clearly in Figure 4c through the relationship between ⬠, ◇, and □. Compare their utilizations with that of ○.
- **Clouds**, like the ones in Figures 4b and 4g, typically characterize samples that are similar in configuration, but separated by noise that offsets points by a small amount in all directions of the PCA space. These clouds can be more or less dense, depending on the coherence of the pattern. The memcache variants, for instance, exhibit strong all-to-all behavior, which manifests as dense clouds to the right of the PCA plots.

Synchronized behavior and noise around a specific configuration capture most of the key behavior in our empirical tests. For these two types of clusters, GMMs are known to perform well. GMMs model a cluster as a multivariate Gaussian with independent parameters for each dimension of the data. This independence provides the flexibility for clusters to fit both types of clusters with arbitrary densities. We fit GMMs to the data using the expectation-maximization algorithm from Scikit-learn [51], which finds clusters that are each defined by a centroid sample and a vector of per-feature variances.

Automated detection of cluster count. GMMs are defined in terms of a fixed number of clusters, K. `tpprof` selects K automatically by using a Bayesian Information Criteria (BIC) score. Informally, a better (lower) BIC score means that a specific clustering, if used as a generative function, is more likely to produce the observed data.

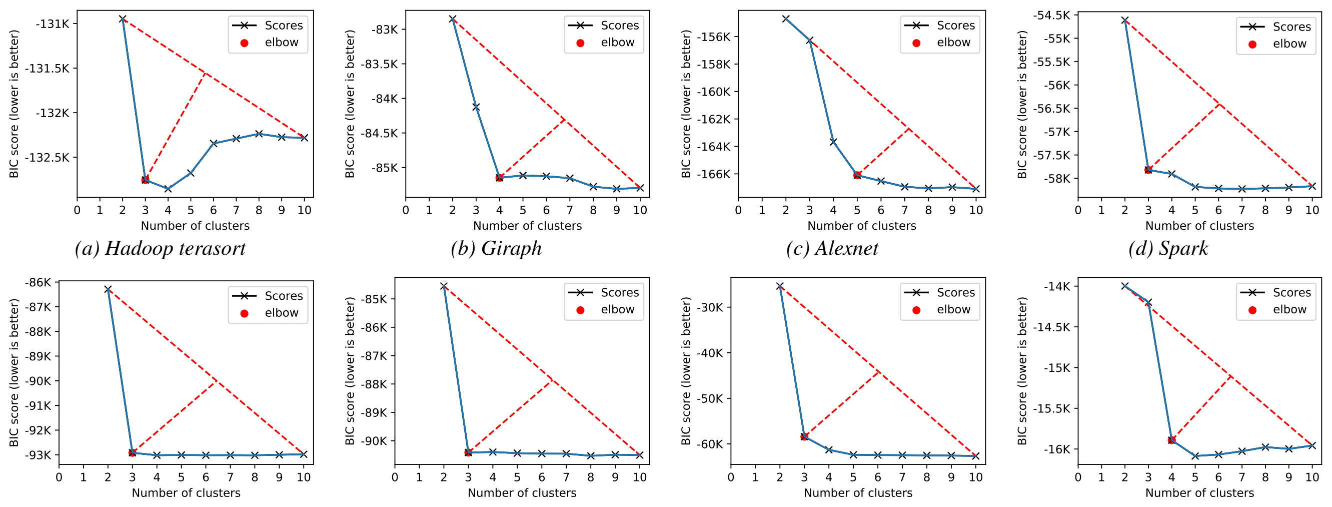
We note, however, that BIC scores tend to improve as K increases, but a high number of clusters can overfit the data. To overcome this issue, we select a K value at which the benefit gained by adding an extra cluster starts to diminish. Finding such "elbows", or points of maximum curvature, is a common problem in machine learning and systems research. We use the Kneedle method [56], a simple but general algorithm based on the intuition that the point of maximum curvature in a convex and decreasing curve is its local minima when rotated θ degrees counter-clockwise about (xmin, ymin) through the line formed by the points (xmin, ymin) and (xmax, ymax). Specifically, we plot the *BIC* score versus K and draw a line segment connecting the points for $K = 2$ and a configured maximum of $K = 10$, which we set based on the typical working set capacity of humans. The optimal value of K is given by the point furthest from that line. Figure 5 shows the results of this analysis for the applications and traces introduced above.

5.2 Network State Subsequences

Network state subsequences extend network states to capture temporal patterns in traffic. Like states, subsequences require compression of the the full sequence of samples taken during the profiling run into a small set of representative patterns. Unlike states, existing sequence-based clustering algorithms are a poor fit for network traffic patterns.

To see why identifying and ranking network state subsequences is challenging, consider a strawman solution: take all possible subsequences of the trace and count their frequencies, e.g., the trace ABC would result in the following subsequences $\{A \times 1, B \times 1, C \times 1, AB \times 1, BC \times 1, ABC \times 1\}$.

Challenge 1: (a) $A^5 = AAAAA$ versus (b) $A^5B \dots AAB \dots AAB$
Intuitively, the interesting bit of sequence (a) is that there is a long run of A's. The strawman solution will instead output that the most common subsequence and frequency is the single state $A \times 5$, followed by $AA \times 4$, etc. With the naïve approach, short subsequences will always take



(a) *Hadoop terasort* (b) *Giraph* (c) *Alexnet* (d) *Spark*
(e) *Memcache* (f) *Memcache imbalance* (g) *Memcache straggler* (h) *Facebook cluster*

Figure 5: Selecting the number of clusters with Bayesian Information Criteria (BIC) and the elbow heuristic.

```
param stateSequence[# samples in the trace]: Full sequence of
  network states.
param minFreq: The minimum number of subsequence
  occurrences before it is counted as a 'common' subsequence.
1  Function getSubSequences:
2      for targetLength : len(stateSequence) to 2 do
3          maxStart ← len(stateSequence) − targetLength
4          for start : 0 to maxStart do
5              end ← start + targetLength
               /* Skip taken ranges */
6              if [start,end] contained in takenRanges then
7                  continue
               /* Add if it meets minFreq */
8              subseq ← log10Merge(stateSequence[start:end])
9              if (# subseq observations) ≥ minFreq then
10                 Add (start, end) to takenRanges after loop
11                 Increment subseqs[subseq]
12             else
13                 Hold subseq until the threshold is reached
14     subsequenceCoverage ← computeCoverage(subseqs)
15     totalCoverage ← computeTotalCoverage(subseqs)
16     return subseqs, subsequenceCoverage, and totalCoverage
```

Figure 6: Pseudocode for finding common subsequences in a sequence of network states.

priority; in fact, we can prove that subsequences will *never* beat their member states. On the other hand, sequence (b) demonstrates a case where it might be useful to be able to observe the shorter subsequences. In this case, greedily setting aside the A^5 would miss the third occurrence of AAB, which is arguably the more important pattern.

Challenge 2: $XA^{39}Y \dots XA^{40}Y \dots XA^{41}Y$
The strawman solution also performs poorly with similar, but not identical ranges. While it may find here that there are long strings of As, or even that X is typically followed by As, or that Y is typically preceeded by As, it will fail to find that As are typically sandwiched between X and Y. Variance in duration is common in networks, where measurement timing, available capacity, and workload size changes frequently.

Challenge 3: $(AB)^3(CDEFGHIJKLMNOPQRSTUVXYZ)^2$
Finally, we note that frequency itself is not an ideal metric. Consider the above trace. The longer trace is much rarer and more interesting, but a pure frequency analysis will rank AB higher in importance.

`tpprof`'s subsequence extraction (outlined in Figure 6) addresses these challenges through a series of rules, which we describe below. Line numbers reference Figure 6.

Only consider subsequences of length 2+ [Line 2]. While knowing the most frequent single states is useful, the goal of extraction is to capture patterns in traffic. We, therefore, prune subsequences of length 1 from consideration and list the relative frequency of single states separately.

Ignore strict subsequences [Lines 6–7]. To better summarize cases like Challenge 1(a), we exclude any subsequence that is wholly contained within another subsequence. We implement this efficiently using two data structures: (1) *takenRanges*, a list of existing subsequences sorted by *start*, and (2) a *heap*-based index of the currently overlapping subsequences, sorted by *end* (not shown).

Frequency threshold before a subsequence is counted [Lines 9–13]. The above rule, applied directly, might produce a single subsequence encompassing the entire trace. To account for this, we set a minimum frequency threshold, *minFreq*, before which the subsequence is not counted. We note that a lower value of *minFreq* promotes the inclusion of longer but sparser subsequences, while a higher value favors many, shorter subsequences. `tpprof` automatically tunes this value using the `hyperopt` library to optimize for 'total coverage', a metric we describe at the end of this section.

Log10 repetition frequencies [Line 8]. To handle cases like

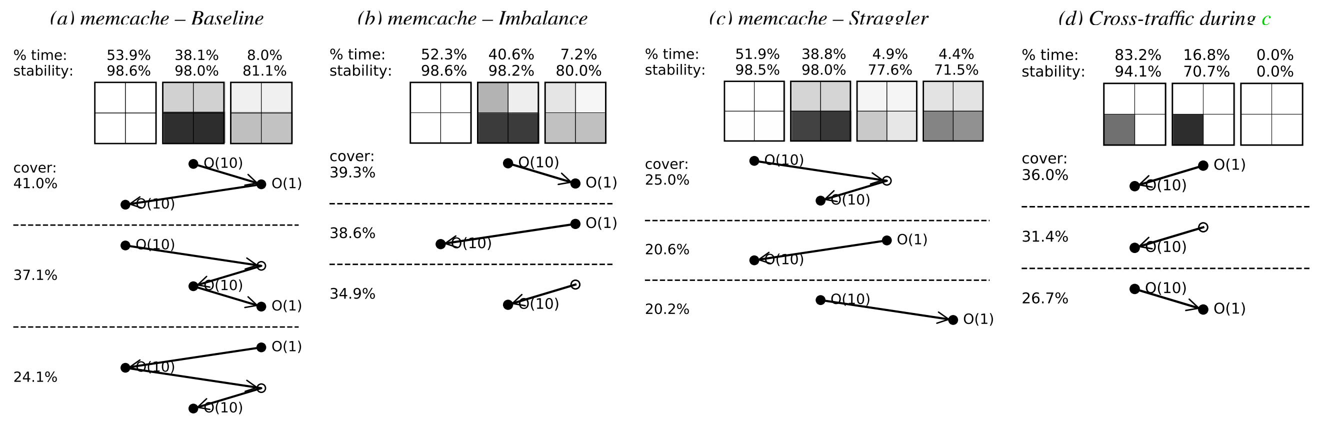


Figure 7: `tpprof` profiles of memcache in three different environments (a–c), plus a profile of cross-traffic (d) active during c.

that of Challenge 2, common in network traces, we compress repetitive states into the nearest power of 10. Doing so ignores small differences in duration while still retaining the length's rough magnitude.

Coverage rather than frequency [Lines 14–15]. As evidenced in Challenge 3, differences in the length of subsequences and the ability of subsequences to overlap diminish the utility of frequency as a way to reason about the relative importance of different subsequences. Instead, we propose *coverage* as a metric for ranking subsequences and for hyperparameter-tuning minFreq. Coverage measures, for either a single subsequence or the union of all subsequences, the cumulative fraction of states in the stateSequence that are included in at least one subsequence.

We encourage the reader to step through several short examples of network state sequences to see why the above rules produce intuitive results.

5.3 Example Visualization: memcached

To tie the above discussion together and showcase the utility of `tpprof`, we present to the reader several real profiles produced by the `tpprof` tool suite. See Section 7 for a description of the hardware testbed used in these tests.

As a baseline, we first look at a memcache workload generated using the `memaslap` [7] benchmarking utility, running in isolation. Each machine in the testbed was configured as a memcache server with 64 B keys and 1024 B values. Gets and sets were randomly generated from two machines—one in each rack—with a ratio of 9:1. In this simple test, the two memcache clients, every 6 s, will simultaneously begin performing 290k get/set operations.We profiled this behavior, collecting a total of 7000 network samples at a 50 ms interval.

Visualization structure. Figure 7a shows the `tpprof` profile for this run. Like Figure 1b, heatmaps of *network state* are at the top of the figure and the most common *network state subsequences* are below. Each heatmap shows the centroid of the sample clusters it represents. We add to this the state's % *time* (the amount of time the network spends in the state) as well as its *stability* (the probability that the network, once in the state, will stay there); states are sorted by % time.

For subsequences, we include the top three by coverage; more can be generated on demand. Subsequences are depicted with a series of points (representing states) connected by arrows (denoting transitions between states). The points align horizontally with the states they represent. Solid points accompanied with an O(x) label indicate an x–$10x$ repetition of that state. The number on the left of each subsequence is the percentage of the trace that it covers. Note that coverage can add to more than 100% due to overlapping subsequences.

Observations. We can observe several characteristics in Figure 7a. First, we can see that there are three states in which the network spends its time. In the one that accounts for more than half the trace, the network is unutilized. The other two show different states of even leaf and even spine utilization, indicating that the network is relatively balanced when it is being used. Note that the leaves of the network are consistently hotter than the spines due to rack-internal communication.

As expected of the workload, the subsequences of the profile show a trace composed of *on-off* periods of *all-to-all* traffic. We can also deduce from the duration of repetitions that the on and off periods both last on the order of seconds. Further, we can infer that the network takes time to ramp up/down from full utilization. This is inferred from the presence of the (L-to-R) 3rd state and the absence of direct transitions between states 1 and 2. Ramp ups seem to be an order of magnitude faster than ramp downs.

`tpprof`'s observations can inform network and application changes. For example, if an operator were to see a similar profile in practice, she could conclude that load balancing is not an issue. Instead, a more promising approach would be to either desynchronize traffic to spread out utilization over time or augment the leaf switches with additional capacity.

5.3.1 Case Study #1: Detecting Load Imbalance

`tpprof` can also help to detect acute problems in networks. As as a case study, we artificially introduced an ECMP miscon-

$\langle signature \rangle$ ::= **{ (**$\langle target\ state\ set \rangle$**) ;** $\langle target\ sequence \rangle$ **}**

$\langle target\ state\ set \rangle$ ::= $\langle target\ state \rangle$ **,** $\langle target\ state\ set \rangle$

$\langle target\ state \rangle$ ::= $\overrightarrow{utilization}$

$\langle target\ sequence\ (P) \rangle$ ::= $\langle target\ state \rangle$ | ~$\langle P \rangle$
  | $\langle P \rangle \wedge \langle P \rangle$ | $\langle P \rangle \vee \langle P \rangle$
  | $\langle P \rangle^*$ | $\langle P \rangle$**{** min repetitions **,** max repetitions**}**

Figure 8: Definition of a traffic pattern signature.

figuration [69] into the network. Specifically, we configured one of the ToR switches to only use the left spine; otherwise, the workload is identical to Figure 7a. Figure 7b shows the output of our tpprof's Python-based visualizer. An operator comparing this profile to that of the baseline would be able to see the new and stark differences between the two spines in all states with load, and conclude that ECMP was not doing its job. While imbalance can also be due to elephant flows and hash collisions, the fact that this happens consistently and always with the same spine points to a structural issue.

5.3.2 Case Study #2: Debugging a Noisy Neighbor

As another case study, we use tpprof to debug an apparent straggler in the system. In this experiment, we add a heavy background flow between two hosts connected to the lower-left leaf, l_1. Figure 7c shows the profile in question. From this profile, an operator can observe that, in 5–10% of samples, there is a slight bias toward l_1 while the other leaf is largely un-utilized. These samples are summarized in the right two network states. If the operator is expecting an even all-to-all pattern like the one in Figure 7a, these states would lead her to suspect that a task in the system is straggling.

tpprof's ability to profile concurrent applications independently can also help to diagnose this issue. In particular, she can view the profile of non-memcache traffic present during the same profiling period. In this case, tpprof would provide her with Figure 7d, which clearly shows a competing flow or set of flows within l_1.

6 Traffic Pattern Scoring

The tpprof components described in prior sections allow users to profile their networks and find prominent traffic patterns. In many cases, after finding certain patterns, users are likely to want to know if (or when) they occur in the future. The tpprof traffic pattern scoring engine solves this problem. The key challenge is designing both a language that makes it simple for users to specify pattern signatures and also an algorithm efficient enough to detect those patterns in realtime.

Traffic pattern signatures. A traffic pattern signature describes the approximate spatial and temporal characteristics of a traffic pattern. It is defined by the grammar in Figure 8 and has two components.

- A set of *target network states* describe the approximate samples that are likely to be observed during the traffic pattern. These can be generated from prior profiling runs or manually specified.
- A *target subsequence*, written as a regular expression, that estimates how the network transitions between the target states during the pattern.

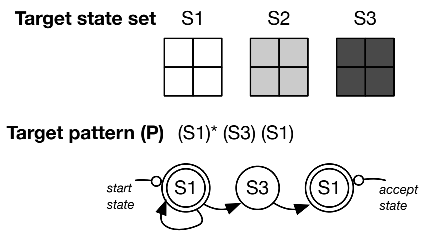


Figure 9: An example traffic pattern signature that detects a synchronized all-to-all burst.

As an example, Figure 9 illustrates a signature to detect a synchronized all-to-all burst of traffic in our example topology. The target states in the signature are: S_1, 0% utilization on all links; S_2, 50% utilization on all links; and S_3, 100% utilization on all links. The signature's target subsequence is, thus, one in which the network is in S_1 before transitioning to S_3 (i.e., high, all-to-all utilization) and immediately going back to S_1, signaling a quick end to the all-to-all utilization.

Scoring signatures. tpprof's Traffic Pattern Score (TPS) algorithm quantifies a signature's prominence in a network sample sequence by finding and scoring subsequences that are similar to it. This amounts to a streaming fuzzy string search. Figure 10 illustrates the scoring algorithm for the all-to-all signature in Figure 9, while Figure 11 provides psuedocode of our streaming implementation. There are three steps.

1. *State matching:* The TPS algorithm first maps each incoming sample to the most similar target state, transforming the stream of samples into an *intermediate stream*.
2. *Pattern matching:* It then scans the intermediate stream for the target subsequence using a finite automata [34]. A match occurs when the automaton reaches an accept state, at which point it is executed in reverse to identify the start point of the longest matching subsequence.
3. *Match scoring:* A match indicates that the exact target subsequence has been found in the intermediate stream; however, how this relates to the underlying sample stream is unclear. Thus, the final step is to score match strength by calculating the average similarity between the two streams during the subsequence.

Writing signatures. There are two sources for signatures. First, they can be automatically generated by the profiler, from the network state subsequences it identifies. This allows the TPS algorithm to automatically identify future reoccurrences of events identified with the tpprof profiler.

Second, users can manually write signatures that characterize the most important attributes of a traffic pattern. Since TPSes use a fuzzy algorithm, patterns do not need to be exact. Instead, they can be defined programmatically. With the

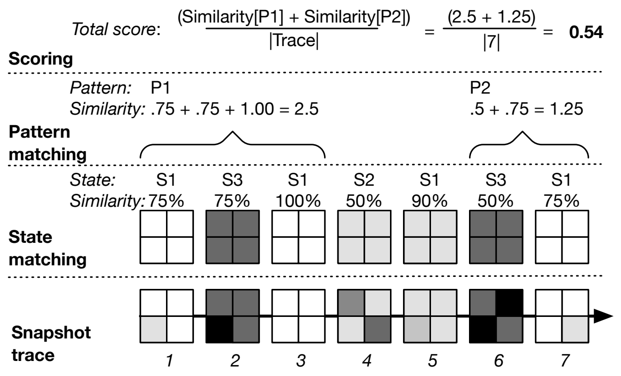

Total score:
(Similarity[P1] + Similarity[P2]) / |Trace| = (2.5 + 1.25) / |7| = 0.54
Scoring
Pattern: P1
Similarity: .75 + .75 + 1.00 = 2.5
P2
.5 + .75 = 1.25
Pattern matching
State: S1 S3 S1 S2 S1 S3 S1
Similarity: 75% 75% 100% 50% 90% 50% 75%
State matching
Snapshot trace
1 2 3 4 5 6 7

Figure 10: Matching and scoring a sample trace against the all-all signature in Figure 9.

```
signature ← (targetStates, regexp)
Function TPSGrep(signature, sampleStream):
    Initialize matchStream
    compile_patterns(matchStream)
    scoreBuf ←[]
    offset ← 0
    for each (sample, timestamp) in trafficPattern do
        /* Identify most similar target state.*/
        stateSymbol ← nearestNeighbor(sample,targetStates)
        similarity ← |netState − sample|
        /* Track scores for up to BUF_LIM of the last samples. */
        scoreBuf.append(score)
        if len(scoreBuf)>BUF_LIM then
            scoreBuf.pop()
            offset ← offset + 1
        /*Invoke HyperScan to update stream. */
        (begin, end) = scan(matchStream, stateSymbol)
        /*If a match occurred, calculate and emit a score. */
        if end ≠ NULL then
            emit sum(scoreBuf[begin-offset:end-offset]
```

Figure 11: The streaming TPS algorithm.

three primitives described below, users can express simple but powerful signatures.

- *State definition*, e.g., `(x:v, y:u)`, which defines a state with switch `x` having utilization `v` and switch `y` having utilization `u`.
- *Set assignment*, e.g., `X:v`. This sets every switch `x` ∈ `X` to utilization value `v`.
- *Iteration (over sets or switches)* e.g., `{(x:v) for x ∈ X}`, which defines a set of states: one state for each switch in `X`, defining that switch to utilization value `v`.

Table 1 lists five example signatures written with these primitives. We evaluate them later in Section 7.3.

7 Implementation and Evaluation

`tpprof` is implemented in Python/C++ as a standalone service that aggregates samples, profiles them, and scores them for the presence of known traffic patterns, as described in the previous sections. Each of the profiles shown in this section are real outputs of `tpprof`, generated programmatically

Pattern	Signature	State Definitions
Short all-all	$S_1^* S_2\{1,10\} S_1$	$\{S_1\}$=`N:0.0`,$\{S_s\}$=N`:0.5`
Long all-all	$S_1^* S_2\{10,100\} S_1$	$\{S_1\}$=`N:0.0`,$\{S_s\}$=N`:0.5`
Hotspots	$(S_1\|S_2\|S_3\|S_4)\{10,100\}$	$\{S_1, ..., S_4\}$=`{(x:1.0, -x:0.0) for x ∈` N`}`
Imbalance	$S_1^*\|S_2^*$	$\{S_1,S_2\}$=`{(x:1.0, -x:0.0) for x ∈ (`s_1`,` s_2`)}`
Stragglers	$(S_1\|S_2\|S_3)^* S_3$	$\{S_1,S_2,S_3\}$=`{(`l_1`:v, -`l_1`:0.0) for v ∈ (0.1,0.01,0.0)}`

Table 1: Traffic pattern signatures for a leaf-spine network N with spines (s_1, s_2) and leaves (l_1, l_2).

using Python and Matplotlib 3.1.1. The requisite counters and polling/batching components that run on each device are implemented in P4 and Python, respectively. Traffic Pattern Scoring is implemented in C++ using `hyperscan` [34].

Hardware testbed. To verify the utility of `tpprof` and its outputs, we used it to profile and score the traffic patterns of real applications running on a small hardware testbed consisting of a Barefoot Wedge100BF-32X programmable switch connected to six servers with Intel(R) Xeon(R) Silver 4110 CPUs via 25 GbE links. The testbed is configured to emulate a small leaf-spine cluster like the one in Figure 1a. To implement this network, we split the Wedge100BF switch into 4 fully isolated logical switches. Each logical switch runs ECMP to balance load across paths.

Application workloads. On our hardware tested, we profile four popular networked applications, in addition to the memcache evaluation in Section 5.3:

1. Hadoop running a TeraSort [11] benchmark workload with 5B rows of data. Our Hadoop instance ran version 2.9.0 with YARN [12] on 10 mappers and 8 reducers spread across the 5 servers (and 1 master).
2. Spark's GraphX [13] running a connected components benchmark workload with 1.24M vertices. We ran Spark 2.2.1 with Yarn on 5 servers (and 1 master).
3. Giraph [10] running a PageRank synthetic benchmark workload with 120,000 vertices and 3,000 edges on each vertex. We used 23 workers in total across our 6 servers.
4. TensorFlow running the AlexNet [38] image processing model with 1 server managing parameters and 5 workers. We used ILSVRC 2012 data for training.

Unless otherwise specified, these applications were run in the presence of background TCP traffic derived from a well-known trace of a large cluster running data mining jobs [8]. Profiles are of of the target application only.

Large-scale trace. To augment our small testbed, we also profile packet traces of 48 Top-of-Rack switches from three of Facebook's production clusters: a frontend cluster, a database cluster, and a Hadoop cluster. As the datasets are sampled by a factor of 30,000, we divide the timestamps by 30,000

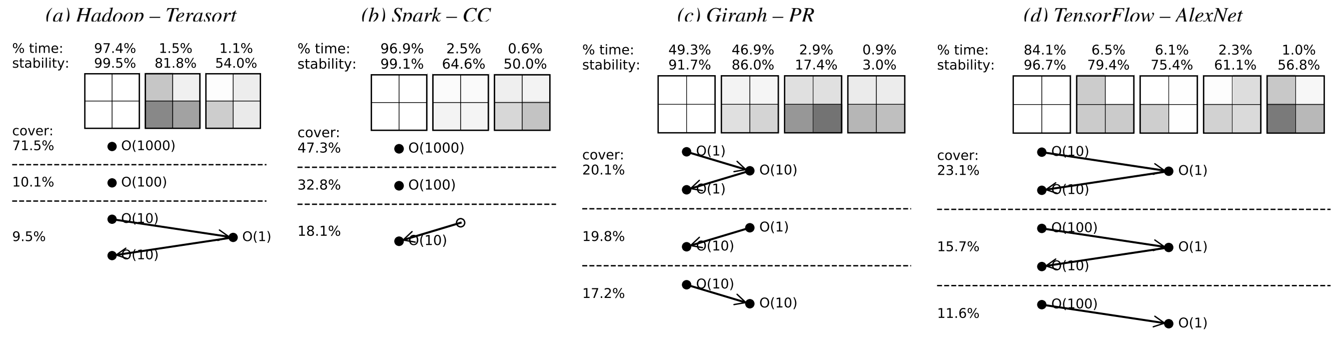


Figure 12: *Profiles of more complex applications running with realistic background traffic.*

to obtain an approximate representation of a full trace. Note that multiplying traffic by 30,000 would have given a more accurate distribution, but resulted in artificially stable patterns.

7.1 Profiling More Complex Applications

To evaluate how `tpprof`'s algorithms deal with more complex applications, we profile each of the application workloads we introduced earlier in this section. These applications all ran in the presence of background traffic, but we only show profiles of the application-specific traffic.

From the resulting profiles in Figure 12, we can see that, for the most part, the network was only lightly utilized during these tests. In Hadoop and Spark, for instance, the network spent $>96\%$ of the time in a unutilized, indicating that our particular testbed tends to be CPU-bound. Giraph is the notable exception, spending about equal time utilized and not.

The states reveal some interesting behavior of the applications. For Hadoop and TensorFlow, we see heavy skew in spine utilization, but not to a consistent spine. This likely indicates the presence of a few large flows that dominate the network and sidestep ECMP's flow-level balancing. We also see in these two workloads a slight bias toward the lower-left switch. This is due to task placement: for Hadoop, that switch is home to the controller and name server; for TensorFlow, it holds both the chief worker and the parameter server.

7.2 Profiling Large Production Networks

`tpprof` is able to profile more complex networks as well. To demonstrate this, we run `tpprof`'s profiler over large-scale traces of the combined traffic for three production Facebook clusters and show the output in Figure 13. We separate the states and subsequences for readability.

Figure 13a shows the profile for the frontend cluster. As in the original paper describing this trace (Figure 5 of [54]), we can observe a clear split between the average utilization of cache, multifeed, and web servers. States A–C show memcache at full utilization, webservers at low utilization, and varying levels of multifeed traffic. Diverging from the original paper, we find an additional network state (occurring 3.8% of the time) in which the multifeed server utilizations spike. The stability of this state indicates that this may manifest as small, but intense and correlated bursts. Subsequences further show frequent transitions between states A and B, with state C representing a short-lived relative lull in multifeed traffic.

Signature	Accuracy	Precision	Recall
Straggler	0.943	0.867	0.720
Imbalance	0.936	1.000	0.868

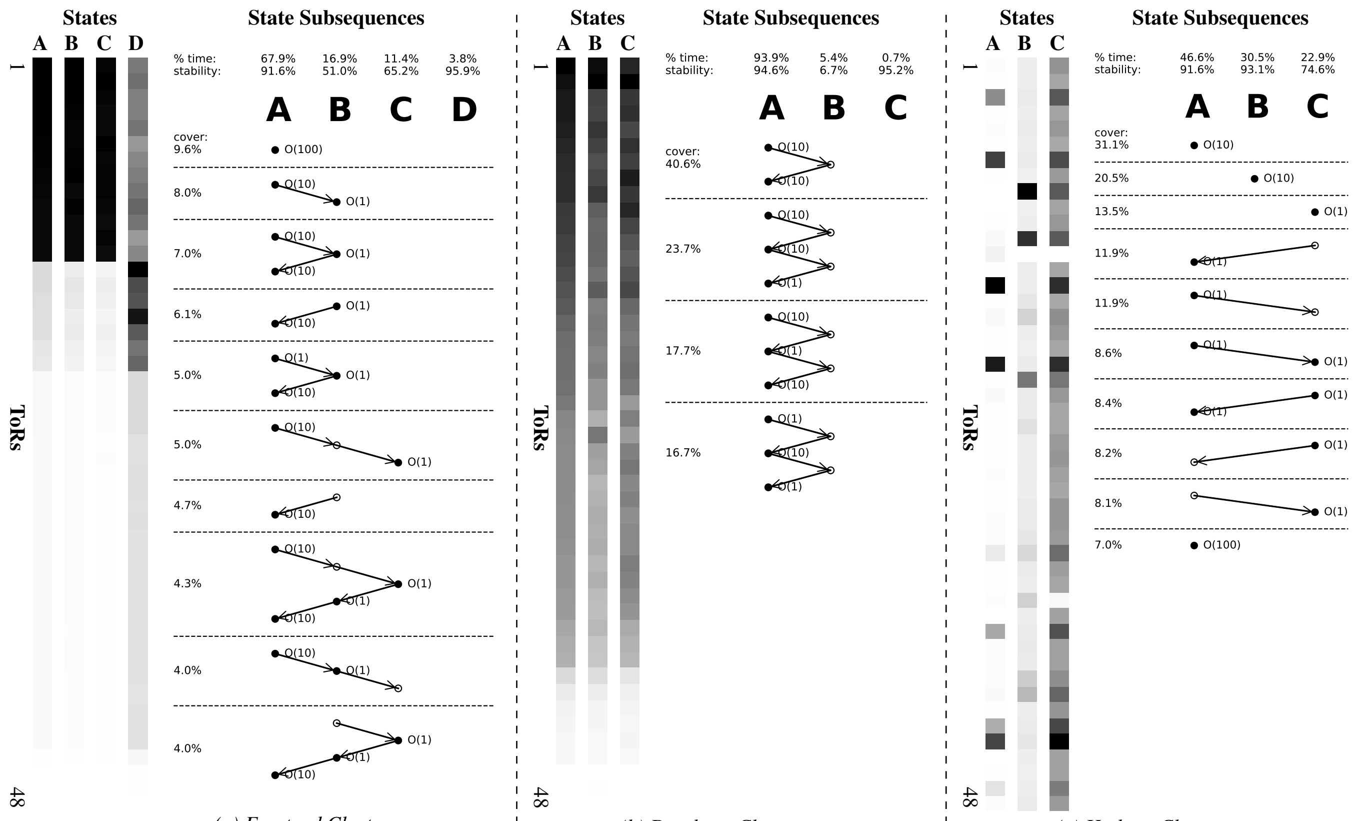
Table 2: *Classification performance of signatures in the memcached testbed.*

Figure 13b and Figure 13c show the profiles of a database and Hadoop cluster, respectively. Notably, the database cluster is very uniform and stable across the trace, indicating a steady workload and good load balancing properties. The Hadoop profile is also notable in that it diverges substantially from the averaged results in Figure 5 of the original paper, which showed balanced utilization across racks. While the traffic is balanced across longer timescales, our results match more closely with their more granular findings of on-off periods and significant variance at medium timescales.

7.3 Efficacy of the TPS Module

We showcase Traffic Pattern Scores by demonstrating how they can help answer an important question: *is my network performing poorly due to load imbalance or stragglers?* For this, we use the straggler and network imbalance signatures from Table 1 to diagnose issues in the memcache deployment from Section 5.3. We run the deployment in baseline, noisy neighbor, and ECMP misconfiguration scenarios. We then generate labeled network sample traces by manually identifying the precise time windows during which each undesired behavior occurred. Finally, we run `tpprof` on each of the traces and compare signature scores against ground truth scores calculated from sample labels.

Figure 14 plots the rolling average of ground truth and signature scores in each of the three scenarios. The signature scores are highly correlated with the ground truth. Table 2 lists the classification performance. Both signatures have high accuracy and precision, with slightly lower recall—a desirable tradeoff in an alerting system. We note that `tpprof`'s per-scenario precision and recall are 100%: no signature's score is high in the baseline scenario; only the straggler sig-



(a) Frontend Cluster (b) Database Cluster (c) Hadoop Cluster

Figure 13: `tpprof` profile of three 48-rack Facebook clusters. Figures include both (1) a collection of states (A–D) organized as a 1×48 heatmap, and (2) a list of the most common state subsequences. Letters map between the two representations.

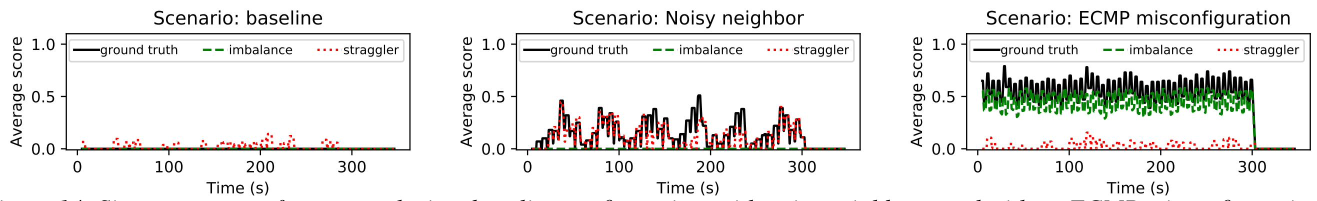


Figure 14: Signature scores for memcache in a baseline configuration, with noisy neighbors, and with an ECMP misconfiguration.

Filter count	0	128	256	512	1024
CPU Util (%)	0.44	0.65	0.84	1.18	1.77

Table 3: `tpprof`'s `iptables` CPU utilization.

nature's score is high in the noisy neighbor scenario; and only the imbalance signature's score is high in the ECMP misconfiguration scenario.

7.4 Overhead and Performance of `tpprof`

Finally, `tpprof` is designed for efficiency and minimal overhead. Only two components in the sampling framework can potentially impact traffic: sample collection and `iptables` tagging. Analytically, snapshots of all ports on a 128 port switch at a 50 ms interval generate only 0.1 Mb/s of measurement data. As Table 3 shows, the `iptables` rules used to construct application-specific profiles also have low overhead.

In addition to measuring overhead, we also benchmark the Hyperscan [34]-based TPS scoring engine, which operates online in parallel with the network. Specifically, we measure average CPU load while operating on the Facebook trace. Figure 15 shows single-core CPU load. It increases linearly with number of signatures, but even in this large network with 100 signatures and a 50 ms sampling frequency, average load for real-time processing is only around 10%.

8 Related Work

Traffic pattern inference. We note that the concept of a network traffic pattern is not novel. Many prior works have both identified and used traffic patterns to great benefit [20, 29, 40, 54, 55, 67]. Unfortunately, these insights have typically been limited to situations were the pattern can be measured at a single link/device [40, 55, 67, 68] or have been a result of property-specific analyses, often with a large dose of manual effort [16, 20, 29, 54]. The goal of `tpprof` is instead

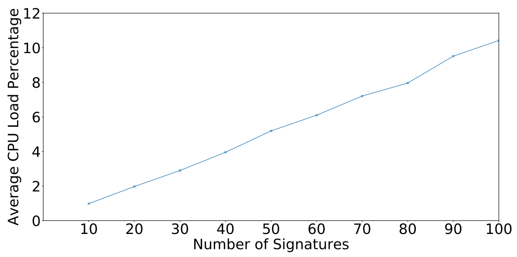


Figure 15: Signature vs CPU Load

the automatic extraction and ranking of common patterns from running networks.

Network monitoring and visualization tools. We also acknowledge the vast array of existing network monitoring and visualization tools, both commercial [1–5, 26, 37, 49, 52] and academic [30, 45, 47, 57, 61, 68]. We lack sufficient space to discuss them all, but one worth mentioning is Cisco's recent Tetration platform [52]. Among other features, Tetration can extract the control flow of a distributed application by clustering hosts based on the partners with which they communicate. Other work has attacked similar problems [36]. To the best of our knowledge, `tpprof` is the first tool that extracts common *network-wide traffic patterns*, rather than application-level communication patterns or packet/flow-level behavior. Broadly speaking, `tpprof` operates at a higher-level of abstraction than these existing systems.

Wee note, however, that `tpprof` is compatible with some infrastructure monitoring frameworks like Nagios [37] that collect monitoring data from across the network. By default, none of these provide the same abstraction as `tpprof`, but many allow custom measurement configurations and plugins, of which `tpprof` could be one.

Application performance profilers. Our work draws inspiration from a long history of work in application performance profiling [19, 22, 27, 46, 53, 60, 64]. Some of which are even able to profile distributed applications [32, 33, 42, 50]. While `tpprof` borrows its approach from the subset of these that profile stochastically, it does this for traffic patterns, which have their own unique set of challenges.

Anomaly detectors. Our alerting mechanisms are related to prior work in anomaly detection. Compared to unsupervised anomaly detection [18, 66], however, `tpprof` provides a much more accurate and fine-grained detection method. Compared to traditional profiling-based anomaly detection in which a user provides a 'correct' trace and the system determines whether the current system diverges [39, 59], `tpprof` can distinguish between different anomalies and does not require the user to obtain a correct trace. More generally, `tpprof`'s scoring engine presents a natural, declarative interface for the user tell the detector, via traffic pattern signatures, the approximate characteristics of relevant traffic patterns.

Clustering and compression. Finally, we note that our techniques for compressing network states borrow from or are related to the rich literature on clustering and compression [14, 23, 25, 35]. Our network state extraction techniques, in particular, leverage existing algorithms. The contribution of this work is instead the choice and tuning of these clustering algorithms to the domain of network traffic pattern analysis.

9 Discussion

Other metrics. While we focus on utilization in this paper, we note that `tpprof` easily extends to any metric collectable from the network. These include simple extensions like packet counts to more advanced metrics like buffer depth and high-water marks. As these metrics are generally correlated with utilization, we anticipate that `tpprof`'s techniques will extend intrinsically, but we leave an exploration of these extensions to future work.

Canned reactions. We also note that the ability of `tpprof`'s scoring engine to distinguish different traffic patterns presents an attractive substrate for building network-level reactions to different traffic patterns. This can also work in reverse: `tpprof` can identify common patterns for which operators should pre-compute reactions. We leave an investigation of this class of applications to future work as well.

10 Conclusion

We present `tpprof`, a network traffic pattern profiler. Just as tools like gprof made it easy for programmers to design, understand, and optimize their programs, `tpprof` does the same for profiling the utilization of large networks. `tpprof` leverages recent advancements in programmable networks and network-wide measurement to capture packet-accurate snapshots of utilization over time. On top of that, `tpprof` builds user-centric profiling, visualization, and automation tools. `tpprof` is agnostic to the application set running over the network and can profile networks in situ, making it an ideal fit for multi-tenant or transit networks. We profile several classic applications in order to demonstrate its utility.

Acknowledgements

We gratefully acknowledge Amin Vahdat for providing comments on drafts, our shepherd Andrew Moore, and the anonymous NSDI reviewers for all of their thoughtful reviews. This work was funded in part by Facebook, VMWare, NSF grant CNS-1845749, and DARPA contract HR0011-17-C0047.

References

[1] https://www.nutanix.com/products/epoch.

[2] https://www.pluribusnetworks.com/.

[3] https://www.logicmonitor.com/.

[4] https://endace.com.

[5] https://www.bigswitch.com/products/big-monitoring-fabric/.

[6] https://aws.amazon.com/blogs/security/tag/network-monitoring-tools/.

[7] http://docs.libmemcached.org/bin/memaslap.html.

[8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[9] Dormando Anatoly Vorobey, Brad Fitzpatrick. Memcached, 2009.

[10] Apache Software Foundation. Giraph, 2012.

[11] Apache Software Foundation. Hadoop, terasort, 2012.

[12] Apache Software Foundation. Hadoop, yarn, 2012.

[13] Apache Software Foundation. Spark, 2016.

[14] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 476–489, New York, NY, USA, 2018. ACM.

[15] Richard E Bellman. *Adaptive control processes: a guided tour*. Princeton university press, 2015.

[16] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, January 2010.

[17] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In Catriel Beeri and Peter Buneman, editors, *Database Theory — ICDT'99*, pages 217–235, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[18] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 93–104, New York, NY, USA, 2000. Association for Computing Machinery.

[19] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[20] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 73–82, New York, NY, USA, 2009. ACM.

[21] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 31–36, New York, NY, USA, 2012. ACM.

[22] W.E. Cohen. Tuning programs with oprofile. *Wide Open Magazine*, 1:53–62, 01 2004.

[23] William HE Day and Herbert Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of classification*, 1(1):7–24, 1984.

[24] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 339–350, New York, NY, USA, 2010. ACM.

[25] Karl Pearson F.R.S. LIII. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[26] Gigamon. Security and networking solutions | gigamon, 2018.

[27] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[28] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 139–152, New York, NY, USA, 2015. ACM.

[29] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 38–49, New York, NY, USA, 2011. ACM.

[30] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation NSDI 14*, pages 71–85, Seattle, WA, 2014. USENIX Association.

[31] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[32] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 95–110, Renton, WA, 2018. USENIX Association.

[33] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed performance monitoring: methods, tools, and applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, June 1994.

[34] Intel. https://www.hyperscan.io/.

[35] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651 – 666, 2010. Award winning papers from the 19th International Conference on Pattern Recognition (ICPR).

[36] Yu Jin, Esam Sharafuddin, and Zhi-Li Zhang. Unveiling core network-wide communication patterns through application traffic activity graph decomposition. In *Proceedings of*

the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '09, page 49–60, New York, NY, USA, 2009. Association for Computing Machinery.

[37] David Josephsen. *Building a Monitoring Infrastructure with Nagios*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[39] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, page 201–206, New York, NY, USA, 2004. Association for Computing Machinery.

[40] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw.*, 2(1):1–15, February 1994.

[41] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[42] Inc. Lightstep. Lightstep [*x*]pm, 2019.

[43] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 399–412, Berkeley, CA, USA, 2013. USENIX Association.

[44] Vincent Liu, Danyang Zhuo, Simon Peter, Arvind Krishnamurthy, and Thomas Anderson. Subways: A case for redundant, inexpensive data center edge links. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 27:1–27:13, New York, NY, USA, 2015. ACM.

[45] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 101–114, New York, NY, USA, 2016. ACM.

[46] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 13–23, New York, NY, USA, 2005. ACM.

[47] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 129–143, New York, NY, USA, 2016. ACM.

[48] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[49] Barefoot Networks. Barefoot deep insight – product brief, 2018.

[50] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[51] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[52] Remi Philippe. Next generation data center flow telemetry. Technical report, Cisco, 2016.

[53] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, NT'97, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[54] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.

[55] Bong K. Ryu and Anwar Elwalid. The importance of long-range dependence of vbr video traffic in atm traffic engineering: Myths and realities. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, pages 3–14, New York, NY, USA, 1996. ACM.

[56] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops*, pages 166–171. IEEE, 2011.

[57] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[58] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM.

[59] Ningombam Anandshree Singh, Khundrakpam Johnson Singh, and Tanmay De. Distributed denial of service attack detection

using naive bayes classifier through info gain feature selection. In *Proceedings of the International Conference on Informatics and Analytics*, ICIA-16, New York, NY, USA, 2016. Association for Computing Machinery.

[60] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 196–205, New York, NY, USA, 1994. ACM.

[61] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 561–575, New York, NY, USA, 2018. ACM.

[62] Nofel Yaseen, John Sonchack, and Vincent Liu. Speedlight bmv2, 2018.

[63] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 402–416, New York, NY, USA, 2018. ACM.

[64] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the mips r10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[65] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[66] Jiong Zhang and Mohammad Zulkernine. Anomaly based network intrusion detection with unsupervised outlier detection. *2006 IEEE International Conference on Communications*, 5:2388–2393, 2006.

[67] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 78–85, New York, NY, USA, 2017. ACM.

[68] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 101–114, New York, NY, USA, 2004. ACM.

[69] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 479–491, New York, NY, USA, 2015. ACM.

TinySDR: Low-Power SDR Platform for Over-the-Air Programmable IoT Testbeds

Mehrdad Hessar[†], *Ali Najafi*[†], *Vikram Iyer and Shyamnath Gollakota*
University of Washington
[†]Co-primary Student Authors

Abstract

Wireless protocol design for IoT networks is an active area of research which has seen significant interest and developments in recent years. The research community is however handicapped by the lack of a flexible, easily deployable platform for prototyping *IoT endpoints* that would allow for ground up protocol development and investigation of how such protocols perform at scale. We introduce tinySDR, the first software-defined radio platform tailored to the needs of power-constrained IoT endpoints. TinySDR provides a standalone, fully programmable low power software-defined radio solution that can be duty cycled for battery operation like a real IoT endpoint, and more importantly, can be programmed over the air to allow for large scale deployment. We present extensive evaluation of our platform showing it consumes as little as 30 uW of power in sleep mode, which is 10,000x lower than existing SDR platforms. We present two case studies by implementing LoRa and BLE beacons on the platform and achieve sensitivities of -126 dBm and -94 dBm respectively while consuming 11% and 3% of the FPGA resources. Finally, using tinySDR, we explore the research question of whether an IoT device can demodulate concurrent LoRa transmissions in real-time, within its power and computing constraints.

1 Introduction

Recent years have seen development of numerous wireless protocols for Internet of Things (IoT) devices. In addition to longtime standards such as Bluetooth and Zigbee, a number of new protocols including LoRa, Sigfox, NB-IoT and LTE-M have been developed that achieve long ranges of more than a few kilometers. Due to the lack of a de-facto standard, this space remains an active area of research for both industry and academia. The rapid advances in this space however present practical challenges for researchers: each of these protocols requires a dedicated radio chipset to evaluate, and these proprietary solutions often leave little room for protocol modification. The academic community is therefore severely handicapped by the lack of a *flexible* platform, as even a complex multi-radio prototype cannot adapt to evaluate new protocols or even customize existing solutions. The current ecosystem therefore discourages researchers from investigating the important questions that arise when scaling up IoT networks, and more importantly taking a systematic approach to developing new protocols from the ground up.

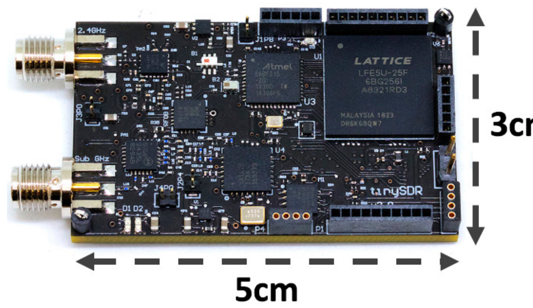


Figure 1: **TinySDR Hardware Platform.** It has two antenna ports for running IoT PHY and MAC protocols at 2.4 GHz and 900 MHz. This image is the actual size of the platform on printed paper.

Ideally, we would like a large scale IoT network testbed with the flexibility to run *any* IoT protocol at the PHY and MAC layers. Further, since many of these IoT testbeds can span hundreds of endpoints across a large campus or even a city, we need the ability to push changes to the PHY and MAC layers, using simple over-the-air software updates. This would allow for performance comparisons on a single testbed to investigate the trade-offs between existing standards as well as showcase the advantages of an entirely new custom protocol. Moreover, to make such a system representative of real-world deployments, individual network nodes should model the constraints of IoT endpoints. Specifically, these devices should have appropriate power controls and options to duty cycle transmissions, have an ultra-low power sleep mode and also have interfaces to connect sensors. Finally, the ability to run these endpoints on batteries would also allow for flexibility of deployment in spaces without dedicated power access, or even in mobile scenarios.

Realizing this vision however is challenging with existing software defined radio (SDR) platforms. Specifically, we require an SDR for the flexibility of implementing differ-

Platform	Sleep Power	Standalone	OTA	Cost	Max BW (MHz)	ADC (bits)	Frequency Spectrum (MHz)	Size (cm)
USRP E310 [7, 17]	2820 mW	✓	✗	$3000	30.72	12	70∼6000	6.8×13.3
USRP B200mini [6, 12]	N/A	✗	✗	$733	30.72	12	70∼6000	5×8.3
bladeRF 2.0 [1, 17]	717 mW	✓	✗	$720	30.72	12	47∼6000	6.3×12.7
LimeSDR Mini [2, 3, 25]	N/A	✗	✗	$159	30.72	12	10∼3500	3.1×6.9
Pluto SDR [18]	N/A	✗	✗	$149	20	12	325∼3800	7.9×11.7
μSDR [9, 10, 30]	320 mW	✓	✗	$150	40	8	2400∼2500	7×14.5
GalioT [5, 63]	350 mW	✓	✗	$60	14.4	8	0.5∼1766	2.5×7
TinySDR	**0.03 mW**	✓	✓	**$55**	**4**	**13**	**389.5∼510, 779∼1020, 2400∼2483**	**3×5**

Table 1: **Comparison Between Different SDR Platforms.** Costs are based on sale prices for commercial products without a public bill of materials (BOM) and published BOM prices for research prototypes. OTA refers to over-the-air programming capabilities.

ent PHY protocols; but there is currently no SDR platform that meets the requirements of IoT endpoints (see Table 1). Existing SDR systems consume large amounts of power for transmitting data, do not support ultra-low power sleep modes, require wired infrastructure and often a dedicated computer and furthermore, are expensive. More importantly, none of the existing SDR platforms support over-the-air programming to update PHY or MAC protocols. Finally, IoT devices prioritize power consumption and communication range and hence use limited radio bandwidth — LoRa, Sigfox, NB-IoT, LTE-M, Bluetooth and ZigBee use only 500 kHz, 200 Hz, 180 kHz, 1.4 MHz, 2 MHz and 2 MHz respectively. In contrast, existing SDR platforms focus on achieving high performance in terms of bandwidth because *they are tailored to the needs of gateway devices and not for IoT endpoint devices.*

Driven by a need for such a platform in our own research, we design tinySDR as shown in Fig. 1, the first SDR platform tailored to the needs of IoT endpoints. TinySDR provides an entirely standalone solution that incorporates a radio front-end, FPGA and microcontroller for custom processing, over-the-air FPGA and microcontroller programming capabilities, a micro SD card interface for storage, ultra-low power sleep modes and highly granular power management options to enable battery-powered operation. It is capable of transmitting and receiving in both the 900 MHz and 2.4 GHz ISM bands, supports 4 MHz of bandwidth which is sufficient for most IoT protocols including Bluetooth, Zigbee, LoRa, Sigfox, NB-IoT and LTE-M, and can achieve the high sensitivities of commercial solutions such as LoRa chips [24]. Additionally it includes multiple analog and digital I/O options for connecting sensors.

Designing such an SDR platform required addressing multiple systems, architecture, power and engineering challenges:

• **Low-power hardware architecture.** Achieving a small form-factor, low-power SDR requires a minimalist design approach that can satisfy the real-time needs of IoT protocols and ensure flexibility at the PHY and MAC layers. To do this, we exploit recent advances in small, low-power microcontrollers, FPGAs and flash memory to pick the right components for our platform (see §3.1). We use a low-power FPGA to run the PHY layer while the microcontroller runs the MAC protocols as well as handles the I/O operations between the FPGA, radio, memory and sensor interfaces (see §3.2).

• **Efficient power management.** Achieving highly granular power management needed for battery-powered operation and enabling ultra-low power sleep modes requires shutting down parts of SDR when not in use. This is important for IoT endpoints that perform duty-cycled operations and require an ultra-low power sleep mode to achieve a long battery life. This presents a design tradeoff between the complexity of toggling the power of each hardware component ON and OFF, and the cost of additional circuitry to do so. We address this challenge in §3.3 and achieve sleep power as low as 30 μW.

• **Over-the-air SDR programming.** Enabling a truly scalable system requires the ability to update the PHY and MAC layers on the platform, over-the-air, in a testbed deployment. This however also introduces the challenge of over-the-air FPGA and microcontroller programming as well as communicating these updates robustly to each device in the network while minimizing power consumption and network utilization. We use a dedicated wireless backbone subsystem complete with a MAC protocol and its own flash memory to program both the microcontroller and FPGA. Additionally we leverage compression and low-power decompression algorithms to minimize network downtime during the updates (see §3.4)

Fig. 2 shows the power consumption of the radio module in tinySDR compared to existing SDR platforms. We evaluate tinySDR's performance by presenting case studies of two common protocols: LoRa and BLE beacons, and also evaluate tinySDR in a campus-testbed of 20 devices.

• LoRa modulation and demodulation use 4% and 11% of the FPGA resources respectively and achieve a sensitivity of -126 dBm for 3.12 kbps, which is similar to an SX1276 [24] LoRa chip with the same configuration. Further, the FPGA supports real-time modulation and demodulation of all LoRa spreading factors from 6 to 12. A LoRa MAC implementation on our MCU is also compatible with the *The Things Network*.

• TinySDR supports 2.4 GHz BLE beacon transmissions. The full baseband packet generation on the FPGA uses 3% of its resources. The platform can perform frequency hopping with a delay of 220 us and achieves a sensitivity of -94 dBm which is comparable to the commercial BLE chipsets [21].

Finally, we present a case study of how the unique capabilities of tinySDR could be used to answer new research questions. Recent work has explored techniques to enable concurrent transmissions in LoRa networks [44, 47]; however these solutions were prototyped on USRPs and it is unclear if IoT endpoints can decode concurrent transmissions in real-time within their power and resource constraints. We implement a custom decoder on tinySDR to demonstrate for the first time that IoT endpoints *can* receive concurrent transmissions.

Contributions. To summarize, we design the first SDR platform tailored to the needs of IoT endpoint devices. By making careful design and architectural choices, our platform achieves low power, supports IoT protocols at both 900 MHz and 2.4 GHz and has computation resources to do on-board processing. We present a highly granular power management scheme that enables duty-cycled operation and 10,000x lower power sleep modes. We also develop the first over-the-air SDR programming capability to support PHY and MAC updates in a wireless testbed. We characterize and evaluate our platform with case studies of LoRa and BLE beacons. Finally, we present a research exploration of concurrently receiving multiple LoRa transmissions on our SDR platform.

Platform availability. TinySDR's hardware schematics and software are available at:

https://github.com/uw-x/tinysdr

2 SDR Requirements for IoT Nodes

To motivate the need for tinySDR and inform our design decisions, we begin by identifying the key requirements for an IoT endpoint. These include 1) operation in the 900 MHz and 2.4 GHz bands, 2) low power operation which requires the ability to transition to ultra-low power sleep mode, 3) standalone operation which requires an on-board control unit to duty cycle the radio, 4) over-the-air programming capabilities for large scale IoT testbeds, 5) low cost per node, and 6) at least 2 MHz bandwidth to support IoT protocols including LoRa, SIGFOX, LTE-M, NB-IoT, ZigBee and Bluetooth. While there are a number of commercially available SDRs such as the USRP, BladeRF, Pluto SDR, and LimeSDR [1, 3, 7, 31, 36] on the market and SDR research prototypes such as WARP, Argos, SORA, SODA, KUAR, Tick, μSDR, OpenMili, and GalioT [40, 41, 43, 46, 55–58, 60, 63, 64, 66–68, 70, 72, 73], all of them are designed as *gateway devices* and do not satisfy many of the above constraints. Here, we analyze the shortcomings of these platforms in the context of these requirements.

• **Low power operation and sleep mode.** Fig. 2 compares the power consumption of the radio module *alone* in existing SDR platforms, since each one has different peripherals. We find that most SDR platforms consume 200-300 mW in receive mode, but a lot more power when transmitting. While this may be acceptable for a gateway devices that are more often receiving, typical IoT endpoints do the opposite and are required to transmit data like sensor information. Moreover, real IoT nodes spend a very short time transmitting before transitioning to ultra-low power sleep modes. Although IoT

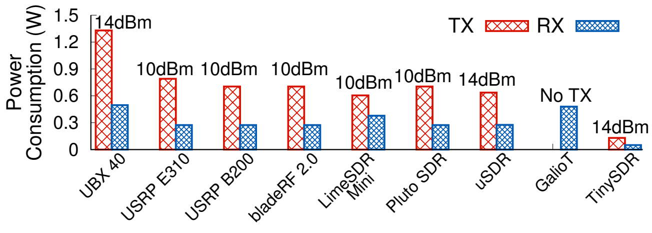


Figure 2: **Radio Module Power Consumption for Each Platform.** The TX output power of each radio module is shown on top of it.

radios often consume tens to hundreds of milliwatts of power, the key to achieving long battery lifetimes is exploiting their microwatt power sleep modes. Table 1 shows that none of the other platforms can benefit from duty cycling as they consume more power in sleep mode than tinySDR does when transmitting; tinySDR's microwatt power consumption in sleep mode enables dramatic power savings with duty cycling.

• **Standalone operation and cost.** We observe that some of these platforms do not allow for standalone operation, i.e., they cannot be used in a testbed deployment without an external computer. Among the ones that do, the Embedded USRP and bladeRF cost $700 or more per unit making large scale deployments expensive. μSDR allows for standalone operation but only operates at 2.4 GHz and cannot support protocols like LoRa. GalioT [63] uses the low cost RTL2832U radio [5] connected to a Raspberry Pi computer which allows for standalone operation, however it does not support 2.4 GHz band. Moreover, this platform is *receiver only* and cannot be used to prototype a typical IoT node that transmits data.

• **Over-the-air (OTA) programming.** As shown in Table 1, all existing SDR platforms rely on wired interfaces for programming. This means that even if one of these systems were connected to a battery, running an experiment would require either tethering each one to a wired network or individually programming them. An OTA programming system is crucial to realizing the goal of a large scale wide area testbed as without it, researchers have to decide between limiting themselves to deployment scenarios with wired infrastructure that are not representative of real IoT use cases or traveling over *kilometer* distances to update individual nodes for *each* minor protocol modification, which would be unmanageable at scale.

3 TinySDR Platform

We first describe our design choices for the different components of our hardware shown in Fig. 3 and explain the interfaces between them. Next we present the power management module which enables our ultra-low-power sleep mode. Finally, we describe our over-the-air update protocol including decompression algorithms and over-the-air reprogramming.

3.1 Hardware Design

We seek to minimize power consumption and cost while offering the flexibility of an SDR to process raw samples.

3.1.1 Designing the Software Radio

The core block on our platform is the software-defined radio, a programmable PHY layer that processes and converts bits to radio signals and vice versa. We begin by explaining our choices for the primary components of an SDR which are a radio chip that provides an interface for sending and receiving raw samples of an RF signal as well as an FPGA that can process these signals in real time. We then discuss the supporting peripherals for these devices such as a power amplifier (PA) to boost the output of the radio chip and non-volatile memory for the FPGA to read and write data from.

Choosing a radio chip. We begin by choosing a radio chip as its specs define the requirements for the FPGA and other blocks. Our primary requirement is that the chip supports reading and writing raw complex I/Q samples of the RF signal. As shown in Table 2, current SDR systems use I/Q radio chips that are designed to cover a multi-GHz spectrum and have high ADC/DAC sampling rates to support large bandwidth. For example, the AD936x [17] series which is used in USRP and Pluto SDR can transmit up to 3.8 GHz and supports sampling rates as high as tens of MHz. Each of these specs such as wide bandwidth, low noise, and high sampling rate represent fundamental trade offs of power for performance, and therefore these chips consume watts of power. Moreover, some of these radio chips costs more than \$100.

We instead take a different approach: identify the minimum required specs and find a radio that supports them. Specifically, a radio chip for an IoT platform must be able to operate in at least the 900 MHz and 2.4 GHz ISM bands, have 4 MHz of bandwidth, while otherwise minimizing power and ideally costing less than \$10. We analyze all of the commercially available radio chips that provide baseband I/Q samples and list them in Table 2, where only the AT86RF215 supports all of our requirements. In addition to its lower cost and support for both frequency bands, it also consumes less power than the MAX2831 and the SX1257. Moreover, the AT86RF215 integrates all the necessary blocks including an LNA, programmable receive gain, automatic gain control (AGC) and low pass filter, ADC on the RX chain, as well as a DAC and programmable PA with a maximum power of 14 dBm on the TX side. In terms of noise, the RF front-end has a 3-5 dB noise figure which is even better than the noise figure of the front-end used in Semtech SX1276 LoRa chipset, suggesting it should be able to achieve long range performance. It consumes 5x less power than the radios used on other SDRs as shown in Fig. 2 and has built in support for common modulations such as MR-FSK, MR-OFDM, MR-O-QPSK and O-QPSK that can save FPGA resources or power by bypassing the FPGA entirely.

Picking an FPGA. Now that we have chosen a radio chip, the next step in our design process is to find an FPGA that can interface with it. Aside from minimizing power and cost, we would also like to maintain a small form factor and short wake-up time. Although flash-based FPGAs are capable of fast wake-ups, they are more expensive compared to SRAM-based FPGAs with the same number of logic elements. We use LFE5U-25F [33] FPGA from Lattice Semiconductor for baseband processing which is SRAM-based and has 24k logic units. This chip provides a greater number of look up tables (LUTs) than the FPGAs on the Pluto SDR and LimeSDR mini, and at a lower cost. Moreover, it is significantly cheaper than the flash-based FPGA used in uSDR [56].

Table 2: **Existing Off-the-Shelf I/Q Radio Modules.**

I/Q Radio	Frequency (MHz)	RX Power (mW)	Cost
AD9361 [17]	70∼6000	262	\$282
AD9363 [18]	325∼3800	262	\$123
AD9364 [12]	70∼6000	262	\$210
LMS7002M [25]	10∼3500	378	\$110
MAX2831 [10]	2400∼2500	276	\$9
SX1257 [35]	862∼1020	54	\$7.5
AT86RF215 [20]	**389.5∼510 779∼1020 2400∼2483**	**50**	**\$5.5**

Adding a power amplifier (PA). AT86RF215 only supports a maximum transmit power of 14 dBm which is traditionally used by IoT radios but is less than the 30 dBm maximum allowed by the FCC. To provide flexibility, we add optional PAs. Given the high cost and power requirements of wide-band PAs that could operate at both 900 MHz and 2.4 GHz we instead select two different chips: the SE2435L [23] for 900 MHz and SKY66112 [28] for 2.4 GHz. Our 900 MHz PA supports up to 30 dBm output power, and the 2.4 GHz PA can output up to 27 dBm. Both chips also include an LNA for receive mode and a built in circuit to bypass either of these components for power savings. In receive mode, we can either pass the incoming signal through the LNA and then connect it to the radio or completely bypass the LNA and connect the signal directly. The maximum bypass current is 280 uA and the sleep current of both power amplifiers is only 1 uA. In transmit operation we can pass the signal through the PA and amplify the signal or turn off the PA and pass the signal directly to the antenna for transmit power $<$ 14 dBm.

Picking the microcontroller. We use a microcontroller to control all the individual chips and toggle all of these power saving options. In addition to having a low sleep current it must be able to support multiple control interfaces, have enough memory resources to support IoT MAC protocols and also be able to run a decompression algorithm for our OTA system. We select the MSP432P401R [27] a 32-Bit Cortex M4F MCU which meets all of our requirements with less than 1 uA sleep current, has 64 KB of onboard SRAM and 256 KB of onboard flash memory. In addition to controlling the I/Q and backbone radio parameters, and reprogramming of the FPGA, the MCU performs the important function of power management. It is responsible for toggling ON and OFF the power amplifiers, as well as performing power-gating by turning ON and OFF different voltage regulators in §3.3.

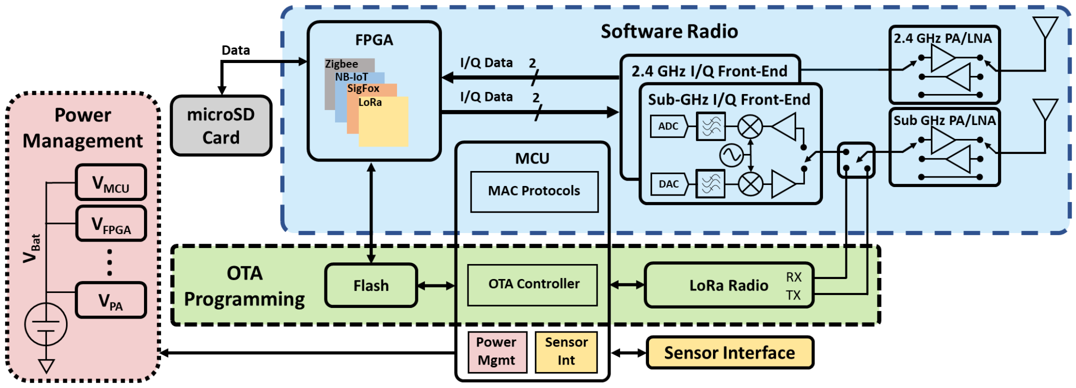


Figure 3: **TinySDR System Block Diagram.** A complete system diagram showing all of the components of tinySDR. This includes the software radio consisting of the radio, amplifiers, and FPGA, OTA programmer which uses a LoRa radio and flash memory to store programs, and a power managment system with the flexibility to turn off power consuming components. Each of these subsystems are controlled in software running on the MCU.

3.1.2 Designing OTA Update Hardware

While the above discussion enables a small, low power, low cost SDR for easy deployment, FPGAs and microcontrollers typically require a wired interface for reprogramming. There are two options for enabling wireless reprogramming: i) using the existing I/Q radio and FPGA and ii) using a dedicated wireless communication chipset. We chose the second option since it provides a fail-safe mode for updating the firmware. Moreover, a dedicated wireless communication chipset would consume less power compared to the first option.

OTA wireless chipset. A key question when designing an OTA update system is, what wireless protocol should be used? To support wide area networking, we focus on protocols designed for long range operation. We analyze all of the available long range protocols and select LoRa for our OTA system for a number of reasons. First, LoRa receivers have a high sensitivity which enables kilometer ranges. LoRa also support a wide range of data rates from 11 bps to 37 kbps which allows us to trade off rate for range depending on the deployment scenario. Moreover, LoRa is becoming more and more wide-spread in the US. We use the SX1276 Semtech chipset [24] which is available for \$4.5, minimizing cost.

Flash Memory. Our FPGA is SRAM based and does not include on-chip non-volatile memory for storing programming data. We instead store the firmware bitstream on a separate flash memory chip. The FPGA programming bitstream is 579 KB and the MCU programs require a maximum of 256 KB. We chose the MX25R6435F flash chip with 8 MB memory. Although this is far more than the size required, it allows tinySDR to store multiple FPGA bitstreams and MCU programs to quickly switch between stored protocols without having to re-send the programming data over the air.

3.2 Interfacing Between Blocks

3.2.1 Reading and Writing I/Q Samples

The AT86RF215 radio chipset samples baseband signals at 4 MHz with a 13 bit resolution for both I and Q. Operating at the full rate therefore requires an interface which can support a throughput of over 100 Mbps without consuming a large amount of power to meet our design objectives. To do this we use low-voltage differential signaling (LVDS) [4] which is a high-speed digital interface that reduces power by using lower voltage signals but maintains good SNR by sending data over two differential lines to reduce common mode noise.

I_SYNC (0b10)	I_Data (13 bits)	Control (1 bit)	Q_SYNC (0b01)	Q_Data (13 bits)	Control (1 bit)

Figure 4: **I/Q Word Structure Used by I/Q Radio.**

Receiving serial I/Q data. Our system communicates over LVDS to the FPGA in serial mode to transfer I/Q data with a physical interface consisting of 4 I/O lines, pairs of which are used to send data and clock signals. The radio outputs 32-bit serial data words at 4 Mwords/s using the format in Fig. 4. Each data word starts with the *I_SYNC* pattern which indicates the start of the *I* sample which we use for synchronization. Next, it has 13 bits of *I_Data* followed by a control bit. The same format follows for *Q*, beginning with a synchronization pattern *Q_SYNC* and then 13 bits for *Q_Data* and the final control bit. The required 128 Mbps data rate is achieved using a 64 MHz clock provided by the radio operating at double data rate by sampling at both the rising and falling edges of the clock. We implement an I/Q deserializer on the FPGA to read the data which samples the input at both the rising and falling edges of the clock, uses the *I_SYNC* and *Q_SYNC* to detect the beginning of the data fields and loads the *I* and *Q* values into 13 bit registers for parallel processing.

Transmitting I/Q samples. In TX mode we need to do the opposite of the above sequence to convert from the parallel representation on the FPGA to a serialized LVDS stream. To do this, we use the FPGA's onboard PLL to generate the 64 MHz clock signal. Next to create our double data rate output signal that varies on both the positive and negative edges of this clock signal using a dual-edge D flip-flop design [48] resulting in the desired 128 Mbps data rate. We use this to generate the same I/Q word structure described above.

3.2.2 Memory Interfaces

After reading the raw data from the LVDS lines using the I/Q deserializer described above, we store the samples into a FIFO buffer implemented using the FPGA's embedded SRAM. We implement a simple memory controller to write data to the FIFO which generates the memory control signals and writes a full data word on each cycle. The embedded memory can run at rates significantly greater than 4 MHz meaning it is not a limiting factor for real-time processing. The SRAM can buffer up to 126 kB. The data stored in the FIFO can then be sent to signal processing blocks to implement filters, cryptographic functions, etc. or to non-volatile flash memory. For flash memory, we use a micro SD card which enables us to collect raw I/Q data and analyze the spectrum. The micro SD card supports two modes: native SD mode and standard SPI mode. In native SD mode, micro SD card's interface uses 4 parallel data lines to read/write data to/from the micro SD card. This mode supports a higher data rate compared to the SPI mode which only supports a 1-bit serial interface. However, we implement SPI mode since it supports the 104 Mbps data rate which we need to write data in real time. This allows us to re-use the same, simpler SPI block for multiple functions and save resources on the FPGA.

3.2.3 RF, Control and Sensor Interfaces

The AT86RF215 provides differential RF signals for both 900 MHz and 2.4 GHz and has an integrated TX/RX switch for both. At 2.4 GHz, the differential signal is transformed to a single-ended output using the 2450FB15A050E [8] balun and fed to the SKY66112 [28] front-end with the bypassable LNA and PA. Finally, after passing through a matching network, the 2.4 GHz signal is connected to an SMA output.

On the 900 MHz side, the differential output of the AT86RF215 is connected to 0896BM15E0025E [32] to convert it to a single-ended output. This must be shared between the backbone radio's two separate RF paths for transmit and receive and AT86RF215's 900 MHz single-ended signal. We choose between them using a ADG904 [19] SP4T RF switch. The single port side is connected to the SE2435L [23] 900 MHz front-end which is similar to the 2.4 GHz front-end. The MCU communicates with the I/Q radio, backbone radio, FPGA and Flash memory through SPI which it uses to send commands for changing the frequency, selecting the outputs, etc. It also has control signals for FPGA programming, 900 MHz and 2.4 GHz front-end modules, RF switch and voltage regulators for active power control. TinySDR supports common digital interfaces like SPI and I2C to communicate with digital sensors and two analog to digital converters (ADC) for interfacing with analog sensors. We leverage the internal flash memory of the MCU ($\approx$ 256 kB) and external flash memory ($\approx$ 8 MB) to store sensor data.

Table 3: **Power Domains in TinySDR.**

Component	Voltage [V]	Power Domain
MCU	1.8V	V1
FPGA	1.1, 1.8, 2.5, Vlvds	V2, V3, V4, V5
I/Q Radio	1.8< V5 <3.6	V5
Backbone Radio	1.8< V5 <3.6	V5
sub-GHz PA	3.5V	V6
2.4 GHz PA	1.8, 3.0	V3, V7
FLASH Memory	1.8	V3
Micro SD Memory	3.0	V7

3.3 Power Management Unit

Next, we present the design of our power management unit which seeks to maximize the system lifetime when running off of a 3.7 V Lithium battery. To enable long battery lifetimes we need to be able to duty-cycle our system and allow the MCU to toggle each of the above blocks ON and OFF when they are not in use. Further, different components have different supply voltage requirements and we wish to provide each one with the lowest voltage possible to minimize power usage.

Ideally we would want separate controllable voltage regulators for each component in the system. However, having many different regulators with individual controls significantly increases the complexity, number of components, and price. Moreover, it complicates the PCB design by requiring many control signals and a multitude of power planes. Therefore, there exists a trade-off between the granularity of power control and the price/complexity of a design. We outline the supply voltages needed for each component and the power domain supporting it in Table 3. Below, we show how we group components to balance power and complexity.

- **Power domain V1 (MCU).** Since the MCU is the central controller that implements power management, it needs to be powered at all times and therefore has its own power domain. To minimize its sleep current we need to use a voltage regulator with a low quiescent current. Although switching voltage regulators have higher conversion efficiency when active, they also have high quiescent currents so we instead select the TPS78218 linear regulator.
- **Power domains V2, V3, V4, V6 and V7.** These power domains provide power to blocks such as the FPGA, memory blocks, and PAs. Since these components can all be turned off when not operating, the voltage regulators for these domains should have low shut-down current during sleep and high efficiency when active. We therefore choose the TPS62240 which has a shutdown current of only 0.1 uA. It is highly efficient and is rated to support the current draw required by all components except the 900 MHz PA. To support this PA at its maximum output power we use the TPS62080 switching regulator which supports the required current.
- **Power domain V5.** V5 is a shared power domain for I/Q radio, backbone LoRa radio and FPGA I/O bank. This power domain is initially set to 1.8V to minimize power consumption, however components such as the radio chips can require higher voltage to achieve maximum output power. Therefore,

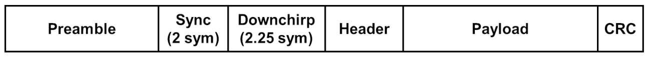


Figure 5: **LoRa Packet Structure.**

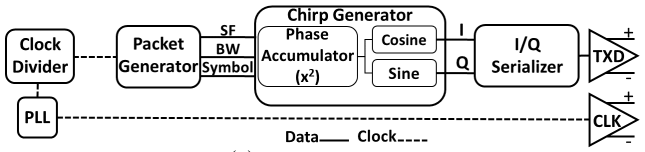
in addition to high efficiency and low shut-down current like the others, this domain should be programmable. To do this, we use Semtech SC195ULTRT [15] which provides an adjustable output that can be set from 1.8 V to 3.6 V.

3.4 Over-the-Air Programming protocol

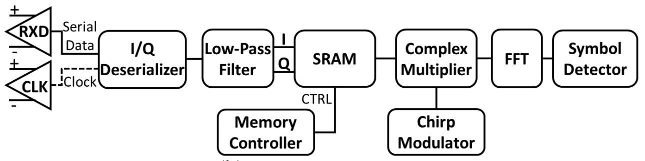
OTA AP and MAC protocol. To update a network of tinySDR devices, we use an AP with a LoRa radio to communicate with each device sequentially. In order to propagate updates throughout a testbed or to specific tinySDR nodes, we design a MAC layer for the LoRa PHY. We pre-program a timer on the MCU to periodically turn off the FPGA and switch from IQ radio mode to the backbone radio to listen for new firmware updates. If there is an update, the AP sends a programming request as a LoRa packet with specific device IDs indicating the nodes to be programmed along with the time they should wake up to receive the update. Upon processing this packet and detecting its ID, the tinySDR node switches into update mode and sends a ready message to the AP at the scheduled time. Then, the AP transmits the firmware update as a series of LoRa packets with sequence numbers. Upon receiving each packet, the tinySDR node checks the sequence number and CRC. For a correct packet it writes the data to its flash memory and transmits an ACK to indicate correct reception. In the case of failure no ACK is sent and the AP re-transmits the corrupted packet after a timeout. After sending all the firmware data, the AP sends a final packet indicating the end of firmware update which tells the tinySDR node to reprogram itself and switch back to normal operation.

To maintain the OTA timing, we use a programmable timer that operates with a 20 PPM low-frequency crystal oscillator source. This will result in timing drift between the tinySDR node and the AP over time. However, with each update we compensate the error by sending the correct time to the tinySDR.

Compressing and decompressing the bitstream. Our system compresses data to reduce update times, however this compression must be compatible with the resources available on tinySDR. We choose the miniLZO compression algorithm [34], which is a lightweight subset of the Lempel–Ziv–Oberhumer (LZO) algorithm. Our implementation of miniLZO only requires a memory allocation equal to the size of the uncompressed data. We perform compression on the AP. The compression ratio of bitstream file varies based on the content of the bitstream, and in the worst case the compressed file could have almost the same size of the original file. This would require a maximum memory allocation of 579 kB which we cannot afford on a low-cost MCU. Instead, we first divide the original update file into blocks of 30 kB that will fit in the MCU memory. Then we compress each



(a) **LoRa Modulator**



(b) **LoRa Demodulator**

Figure 6: **LoRa Implementation Block Diagrams.**

block separately and transmit them to the tinySDR node one by one. Considering the LoRa radio takes more power than the MCU, we immediately write the data to our dedicated programming flash memory using an SPI interface.

After receiving all the data we turn off the LoRa radio and decompress data. First, we allocate memory on the MCU's SRAM equal to the block size and load a block of data from flash. Next, we perform decompression and write the data in the allocated SRAM memory. Finally, we write the decompressed data back to the flash beginning at the corresponding address of the programming boot file. We repeat these steps until we decompress the full firmware update.

Over-the-air FPGA programming. After storing uncompressed programming data in flash memory, we program the FPGA. We use the MCU to set the FPGA into programming mode. When the FPGA switches to programming mode, it automatically reads its firmware directly from the flash memory using a 62 MHz quad SPI interface and programs itself. Reading from flash using quad SPI achieves programming times of 22 ms which is similar to FPGAs with embedded flash memory and results in minimal system down time. After programming is complete, it resumes operation and begins running the new firmware.

3.5 TinySDR's Architectural Considerations

We design tinySDR to achieve three main goals: i) low-power ii) low-cost and iii) over-the-air programmability. To do this, we use a low-power I/Q radio with lower bandwidth support compared to previous platforms. Since this radio is optimized for low bandwidths and in turn low sampling rates, it consumes less power during TX/RX operations. Previous platforms [31, 56] use hardware architectures that support high-bandwidth protocols such as Wi-Fi. However, we use a low-power radio and build our hardware architecture for IoT protocols around it. In addition, we design a power management system to be able to power cycle different parts of tinySDR's architecture in each operation to further reduce the power consumption. To achieve this, we use an MCU chip that enables full control of the tinySDR's blocks and power domains. We achieve minimum power consumption during sleep mode by turning off power-hungry components.

In contrast, previous architectures such as uSDR [56] use an MCU integrated with an FPGA which forces the FPGA to be always ON and increases the power consumption during sleep mode. This is because uSDR is not designed for IoT endpoints but for gateways and hence does not provide any of the architectural optimizations required for the low-power sleep operation required by IoT devices. Furthermore, by using a lower bandwidth radio and also a low-cost SRAM-based FPGA, we minimize the cost compared to platforms such as uSDR [56], Pluto SDR [31] and LimeSDR [2]. Finally, we design tinySDR to operate completely standalone on battery without the need for a wired connection to a network or a computer. To do this, we design an OTA system on tinySDR to be able to re-program the FPGA and MCU on tinySDR wirelessly. This capability does not exist on prior SDR platforms.

4 Case Studies: LoRa and BLE Beacons

4.1 LoRa Protocol with tinySDR

We choose LoRa as it is gaining popularity for IoT solutions due to its long range capabilities. Since LoRa is a proprietary standard, we begin by describing the basics of its modulation and packet structure followed by the implementation details of our modulator, demodulator and MAC protocol.

LoRa Protocol Primer. LoRa achieves long ranges by using Chirp Spread Spectrum (CSS) modulation. In CSS, data is modulated using linearly increasing frequency upchirp symbol. Each upchirp symbol has two main features: Spreading Factor (SF) and Bandwidth (BW). SF determines the number of bits in each upchirp symbol [44, 47, 69] and BW is the difference between upper and lower frequency of the chirp which together with SF determines the length of an upchirp symbol. SF and BW trade data rate for range. Data is modulated by 2^{SF} cyclic-shifts of an upchirp symbol. The starting point of the symbol in frequency domain, which is the cyclic shift of the upchirp symbol, determines its value [16]. LoRa uses SF values from 6 to 12 and BW values from 7.8125 KHz to 500 KHz to achieve PHY-layer rates of $\frac{BW}{2^{SF}} \times SF$.

Fig. 5 shows the LoRa packet structure which begins with a preamble of 10 zero symbols (upchirps with zero cyclic-shift). This is followed by the Sync field with two upchirp symbols. Next, a sequence of 2.25 downchirp symbols (chirp symbol with linearly decreasing frequency) indicate the beginning of the payload. The payload then consists of a sequence of upchirp symbols which encode a header, payload and CRC.

LoRa Modulator. Fig. 6a shows the block diagram of our LoRa modulator. We use our FPGA to implement a LoRa modulator in Verilog and stream data to AT86RF215 in I/Q mode. The modulator begins with the *Packet Generator* module which reads data either from FPGA memory for transmitting fixed packets or from the MCU, as well as LoRa

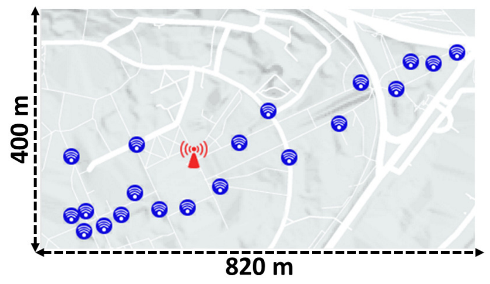


Figure 7: **Evaluation Testbed Map.**

configuration parameters such as SF, coding and BW. This module determines each symbol value and its corresponding cyclic-shift. Next, the *Packet Generator* sends these parameters along with the symbol values to the *Chirp Generator* module, which generates the I/Q samples of each chirp symbol in the packet using a squared phase accumulator and two lookup tables for *sine* and *cosine* function [69]. We then feed these I/Q samples into an I/Q Serializer to stream them over the LVDS interface to the I/Q radio. We generate the 64 MHz transmission clock using internal PLL of the FPGA.

LoRa Demodulator. Fig. 6b shows the block diagram of our LoRa demodulator. It begins by reading data from the I/Q radio into the *I/Q Deserializer* module on the FPGA which converts the serial I/Q stream to parallel I/Q for further signal processing. Next, we run the data through a 14 tap FIR low-pass filter to suppress high frequency noise and interference. We store the filtered samples in a buffer implemented using the FPGA's memory blocks. To decode the data, we use the *Chirp Generator* module from the *LoRa Modulator* described above to generate a baseline upchirp/downchirp symbol, and then we multiply that with the received chirp symbol using our *Complex Multiplier* unit. The output of the multiplication then goes to an FFT block implemented using a standard IP core from Lattice. Finally the *Symbol Detector* scans the output of the FFT for peaks and records the frequency of the peak to determine the symbol value. To detect the chirp type (upchirp/downchirp), we multiply each chirp symbol with both an upchirp and downchirp and then compare the amplitudes of their FFT peaks. The higher peak in the FFT shows higher correlation which indicates the chirp type.

LoRa MAC Layer. To demonstrate that our LoRa implementation on tinySDR is compatible with existing LoRa networks such as the LoRa Alliance's [26] The Things Network (TTN) [38], we adopt their LoRa MAC design from TTN's Arduino libraries [39] and implement it on tinySDR's MCU. TTN uses two methods for device association; Over-the-air activation (OTAA) and activation by personalization (ABP). In OTAA, each node performs a join-procedure during which a dynamic device address is assigned to a node. However, in ABP we can hard-code the device address in the device which makes it simpler since the node skips the join procedure. Our platform can support both OTAA and ABP methods.

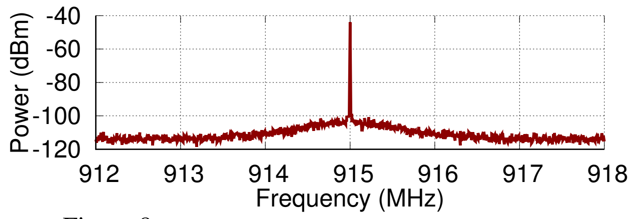


Figure 8: **TinySDR Single-Tone Frequency Spectrum.**

Table 4: **Different Operation Timing for TinySDR.**

Operation	Duration (ms)
Sleep to Radio Operation	22
Radio Setup	1.2
TX to RX	0.045
RX to TX	0.011
Frequency Switch	0.220

4.2 BLE Beacons with tinySDR

To demonstrate tinySDR's 2.4 GHz capabilities we implement Bluetooth beacons which are commonly used by IoT devices.

BLE Beacon Primer. We implement non-connectable BLE advertisements (ADV_NON_CONN_IND) which are broadcast packets used for beacons. These packets allow a low power device to broadcast its data to any listening receiver within range without the power overhead of exchanging packets to setup a connection. These packets have a bit rate of 1 Mbps in Bluetooth 4.0 or up to 2 Mbps in Bluetooth 5.0 and are generated using GFSK with a modulation index of 0.45-0.55. The GFSK modulation is binary frequency shift keying (BFSK) with the addition of a Gaussian filter to the square wave pulses to reduce the spectral width.

Generating a BLE Packet. Bluetooth advertisements consist of 6-37 octets, beginning with fixed preamble and access address fields indicating the packet type set to 0xAA and 0x8E89BED6 respectively. This is followed by the packet data unit (PDU) beginning with a 2 byte length field and followed by a manufacturer specific advertisement address and data. The final 3 bytes of the packet consist of a CRC generated using a 24-bit linear feedback shift register (LFSR) with the polynomial $x^{24}+x^{10}+x^9+x^6+x^4+x^3+x+1$. The LFSR is set to a starting state of 0x555555 and the PDU is input LSB first. The final LFSR state after inputting the PDU becomes the CRC. Data whitening is then performed over the PDU and CRC fields to eliminate long strings of zeros or ones within a packet. This is also done using a 7-bit LFSR with polynomial x^7+x^4+1. The LFSR is initialized with the lower 7 bits of the channel number the packet will be transmitted on, and each byte is input LSB first. We implement both these blocks in Verilog on the FPGA.

Packet Transmission and MAC Protocol. From this bitstream, we need to generate the I/Q samples to feed to the I/Q radio. First, we upsample and apply a Gaussian filter to the bitstream. This gives us the desired changes in frequency which we integrate to get the phase. We then feed the phase to *sine* and *cosine* functions to get the final I and Q samples, which are passed to I/Q serializer and sent to the I/Q radio. BLE divides the 2.4 GHz band into channels, each spaced 2 MHz apart, but BLE beacons are only transmitted on three advertising channels without carrier sense, typically in sequential order separated by a few hundred microseconds. This sequence is re-transmitted every advertising interval [37].

5 Evaluation

We deploy a testbed of 20 tinySDR devices across our institution's campus as shown in Fig. 7. To see if tinySDR meets the requirements for IoT endpoint devices, we characterize its power, computational resource usage, delays and cost when operating in different modes and running different protocols.

5.1 Benchmarks and Specifications

Sleep mode power. Many IoT nodes perform short, simple tasks allowing them to be heavily duty cycled which allows them to achieve battery lifetimes of years. We design tinySDR with this critical need in mind such that the MCU can actively toggle on and off power consuming components such as the radio, PAs, and FPGA to enter a low power sleep mode.

We do this by first turning off the the I/Q transceiver and LoRa radios. To reduce the static power consumption of the FPGA, we shut it down by disabling the voltage regulators that provide power to its I/O banks and core voltage. Similarly, we also turn off the PAs. Finally, we put the MCU in sleep mode LPM3 running only a wakeup timer. The measured total system sleep power in this mode was 30 uW.

The low sleep power allows for significant power savings, but also introduces latency. Table 4 shows the time required to wake up from sleep mode until the radio is active. Because we can perform the I/Q radio setup in parallel with booting the FPGA, the total wakeup time for RX and TX is 22 ms. The I/Q radio setup takes 1.2 ms, so the wakeup time is dominated by booting up the FPGA which itself takes 22 ms. We compare this to a SmartSense Temperature sensor [14] and find that tinySDR has only a 4x longer wakeup time even though it requires programming unlike commercial products that use a custom single protocol radio. Additionally many IoT devices operate at low duty cycles waiting in sleep mode for seconds or more making tinySDR's wakeup latency insignificant.

Switching delays. We also measure the switching delays for different operations on the I/Q radio as this is an important parameter for meeting MAC and protocol timing requirements. Table 4 shows that it takes 45 μs and 11 μs to switch from TX to RX mode and RX to TX mode respectively. As we see later, this is sufficient to meet the timing requirements of IoT packet ACKs and MAC protocols. Further, the delay for switching between different frequencies is only 220 us. To measure this number, we switch between 2.402 GHz, 2.426 GHz and 2.480 GHz. This switching delay is again sufficient to meet the requirements of frequency

Table 5: **TinySDR Cost Breakdown for 1000 Units.**

Components		Price
DSP	FPGA	$8.69
	Oscillator	$0.9
IQ Front-End	Radio	$5.08
	Crystal	$0.53
	2.4 GHz Balun	$0.36
	Sub-GHz Balun	$0.3
Backbone	Radio	$4.5
	Crystal	$0.4
	Flash Memory	$1.6
MAC	MCU	$3.89
	Crystals	$0.68
RF	Switch	$3.14
	Sub-GHz PA	$1.54
	2.4 GHz PA	$1.72
Power Management	Regulators	$3.7
Supporting Components	–	$4.5
Production	Fabrication [22]	$3
	Assembly [22]	$10
Total	–	**$54.53**

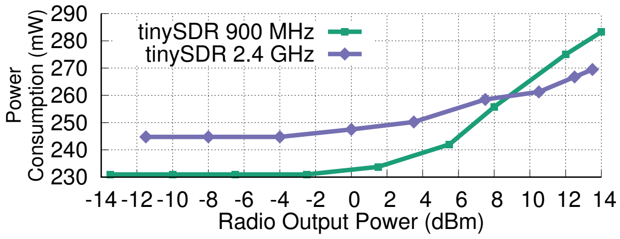


Figure 9: **Single-Tone Transmitter Power Consumption.** We show the total power consumption of tinySDR including I/Q radio, FPGA, MCU and regulators at different transmitter output power. This is 15-16 times lower power consumption than the USRP E310 embedded SDR.

hopping during Bluetooth advertising.

Transmitter performance. First, we implement a single-tone modulator on the FPGA that generates the appropriate I/Q samples and streams them over LVDS to the radio. We connect the output to an MDO4104b-6 [11] spectrum analyzer and observe a single tone, shown in Fig. 8, with no unexpected harmonics introduced by the modulator.

Next we measure the end-to-end DC power consumption of our system including the I/Q radio, FPGA, MCU and regulators to see how it scales with RF output power. We vary our radio output power while transmitting a single tone and use a Fluke 287 multimeter to measure its DC power draw. Fig. 9 shows the power consumption of tinySDR for 900 MHz and 2.4 GHz operation. Interestingly, we observe the DC power is constant at low RF power but increases as expected beyond some RF power level. TinySDR consumes 231 mW when transmitting at 0 dBm, and for comparison the end-to-end power consumption of the USRP E310 is 16x higher under the same conditions. Similarly tinySDR consumes 283 mW at its 14 dBm setting while the USRP E310 is 15x higher. In addition, we measure the peak current consumption of tinySDR while transmitting a single-tone on the I/Q radio. The peak current consumption is 105 mA when tinySDR boots up the FPGA and then starts transmitting using the I/Q radio. This current is less than the maximum current supported by a typical LiPo 1200 mAh battery [13].

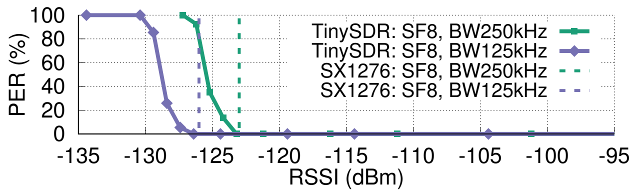


Figure 10: **LoRa Modulator Evaluation.** We evaluate our LoRa modulator in comparison with Semtech LoRa chip.

Cost. We also analyze the cost which is an important practical consideration for real world deployment at scale. Table 5 shows a detailed breakdown of cost including each component as well as PCB fabrication and assembly based on quotes for 1000 units [22], where the overall cost is around $55 each.

5.2 Evaluating the Case Studies

LoRa using tinySDR. We evaluate various different components of tinySDR using LoRa as a case study.

LoRa modulator. To evaluate this, we use our LoRa modulator to generate packets with three byte payloads using a spreading factor of $SF = 8$ and bandwidths of 250 kHz and 125 kHz which we transmit at -13 dBm. We receive the output of tinySDR on a Semtech SX1276 LoRa transceiver [35] which we use to measure the packet error rate (PER) versus RSSI and plot the results in Fig. 10. We compare our LoRa modulator to transmissions from an SX1276 LoRa transceiver. The plots show that we can achieve a comparable sensitivity of -126 dBm which is the LoRa sensitivity for $SF = 8$ and $BW = 125kHz$ configuration. This is true for both configurations, which shows that our low-power SDR can meet the sensitivity requirement of LPWAN IoT protocols.

LoRa demodulator. Next we evaluate our LoRa demodulator on tinySDR. To test this, we use transmissions from a Semtech SX1276 LoRa transceiver and use tinySDR to receive these transmissions. The LoRa transceiver transmits packets with two configurations using a spreading factor of 8 and bandwidths of 250 kHz and 125 kHz. We record the received RF signals in the FPGA memory and run them through our demodulator to compute a chirp symbol error rate. Note that the Semtech LoRa transceiver does not give access to its symbol error rate, but since we have access to I/Q samples, we can compute it on our platform. We plot the results in Fig. 11 as a function of the LoRa RSSI values. Our LoRa demodulator can demodulate chirp symbols down to -126 dBm which is LoRa protocol sensitivity at $SF = 8$ and $BW = 125kHz$. Both the LoRa modulator and demodulator run in real-time.

Resource allocation. Next, we evaluate the resource utilization of our LoRa PHY implementation on the FPGA. Table 6 shows the size for implementing the modulator and demodulator on our FPGA using different SFs. Our LoRa modulator supports all LoRa configurations with different SF with no additional cost. However, in the LoRa demodulator, we need

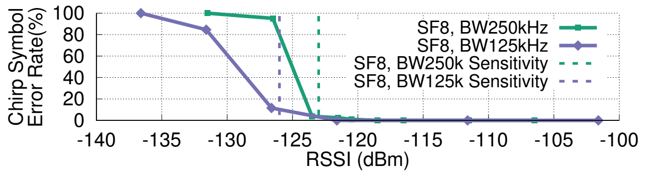


Figure 11: **LoRa Demodulator Evaluation.** We evaluate our LoRa demodulator by demodulating chirp symbols at different RSSI.

Table 6: **FPGA Utilization for LoRa Protocol.**

SF	LoRa TX (LUT)	LoRa RX (LUT)
6	976 (4%)	2656 (10%)
7	976 (4%)	2670 (10%)
8	976 (4%)	2700 (11%)
9	976 (4%)	2742 (11%)
10	976 (4%)	2786 (11%)
11	976 (4%)	2794 (11%)
12	976 (4%)	2818 (11%)

FFT blocks with different sizes to support different SF configurations. This table shows that our FPGA has sufficient resources to support multiple configurations of LoRa and still leave space for other custom operations.

LoRa MAC. We implement the LoRa MAC based on TTN's Arduino libraries [39]. TTN protocol together with control for the I/Q radio, backbone radio, FPGA, PMU and decompression algorithm for OTA take only 18% of MCU resources. Also, as shown in Table 4, our timings are well within the requirements for LoRaWAN specifications [26].

We also measure the power consumption of our platform for LoRa packet transmission and reception. LoRa packet transmission with $SF = 9$ and $BW = 500\ kHz$ and radio output power of 14 dBm consumes a total power of 287 mW from which 179 mW is for the radio and the rest is from the FPGA and MCU. LoRa packet reception consumes 186 mW with radio taking 59 mW.

BLE using tinySDR. Next, we evaluate tinySDR using BLE beacons as a case study. First, we measure the impact of our BLE beacons transmitted from tinySDR using the TI CC2650 [21] BLE chip as a receiver. We do this by configuring tinySDR to transmit BLE beacons at a rate of 1 packet per second. We transmit 100 packets and set the CC2650 BLE chip to report bit error rate (BER). Fig. 12 shows the BER as a function of the received RSSI as reported by the CC2650 BLE chip. The plot shows that we achieve a sensitivity of -94 dBm. This is within 2 dB of the CC2650 BLE chipset's sensitivity, defined by a BER threshold of 10^{-3}.

Next we evaluate the latency of our BLE implementation as BLE beacons are typically transmitted in sequence by hopping between three different advertising channels. We measure the minimum time tinySDR takes to switch between these frequencies by connecting its output to a 2.4 GHz envelope detector and using an MDO4104B-6 oscilloscope to measure the time delay between transmissions. Fig. 13 plots the envelope of three BLE beacons in the time-domain transmitted on the different advertising channels and shows that our system can transmit packets with as little as 220 us delay

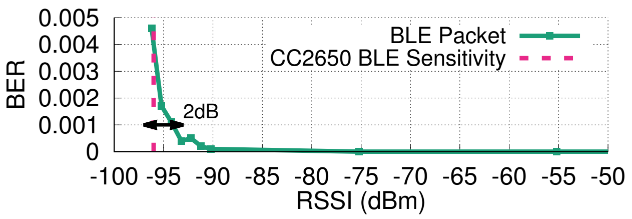


Figure 12: **BLE evaluation.** BLE beacons at different power levels.

between beacons. The corresponding result when a iPhone 8 transmits beacons is 350 us. Finally, generating BLE beacons requires only 3% of the FPGA resources on the tinySDR and it could run for over 2 years on a 1000 mAh battery when transmitting once per second.

5.3 Over-the-Air Programming

An effective OTA programming system should both minimize use of system resources such as power as well as network downtime. Considering the time to reprogram the FPGA and microcontroller from flash is fixed, the downtime for programming a node depends on the amount of data sent and the throughput which varies with SNR.

Raw programming files for our FPGA are 579 kB, however we compress our data using miniLZO. While the exact compression ratio depends on FPGA utilization, our LoRa program compresses to 99 kB and BLE to 40 kB. Our microcontroller programs for both LoRa and BLE are approximately 78 kB and are both compressed to 24 kB. When dividing the files into packets, we would ideally minimize the preamble length and maximize packet length to reduce overhead, however long packets with short preambles lead to higher PER. We choose a preamble of 8 chirps and packets of 60 B which we find balances the trade-off of protocol overhead versus range in our experiments.

To see the impact on a real deployment, we evaluate the time required to program tinySDR nodes in our 20 device testbed shown in Fig. 7. We set up a LoRa transceiver configured with $SF = 8$, $BW = 500\ kHz$ and $CodingRate = 6$ connected to a patch antenna transmitting at 14 dBm as an AP and measure the time it takes to program the tinySDR devices at each location, according to our protocol. We transmit the compressed FPGA and MCU programming data for LoRa and BLE and plot the results as a CDF in Fig. 14. The plots show that the LoRa FPGA requires an average programming time of 150 s while BLE, FPGA, and MCU require 59 s and 39 s respectively due to their smaller file size. Decompressing these received files only takes a maximum of 450 ms. The variation of the programming time between different nodes is caused by the variation in the wireless channel and hence the number of re-transmissions for each tinySDR node is different.

Our OTA programming system components, backbone radio and MCU, consume an average energy of 6144 mJ for receiving a LoRa FPGA update and 2342 mJ for a BLE FPGA update when using 14 dBm output power. Using a 1000 mAh

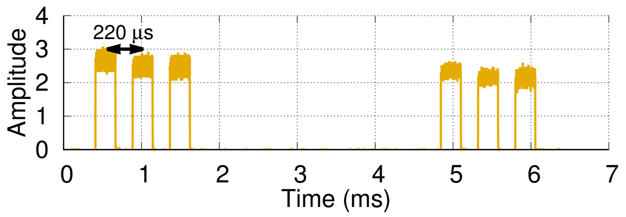


Figure 13: **BLE Beacons Signal.** We show BLE beacon transmissions on three advertising channels from tinySDR using an envelope detector.

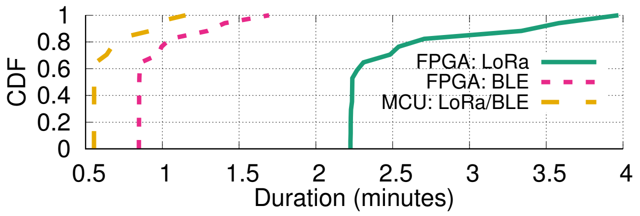


Figure 14: **OTA Programming Time.** We show CDF of OTA programming time for programming LoRa and BLE implementations on tinySDR.

LiPo battery, we could OTA program each tinySDR node with LoRa 2100 times and BLE 5600 times. Assuming OTA programming of once per day, the average power consumption would be 71 uW and 27 uW respectively for LoRa and BLE.

6 Research Study: Concurrent Reception

An SDR designed for IoT endpoints that can provide I/Q transmission and reception capability opens up opportunities for addressing multiple research questions in IoT networks.

In this section, we focus on the following question: Can a *low-power IoT endpoint* device decode multiple concurrent LoRa transmissions at the same time? LoRa supports long range communication for IoT devices and is gaining popularity as a low-power wide area networking (LPWAN) standard. Supporting long ranges introduces new challenges since it increases the probability of collisions in large scale city-wide deployments. While recent works [44, 47] have explored the feasibility of enabling concurrent LoRa transmissions, they have been designed for decoding on a gateway-style USRP device. In fact, most concurrent transmission techniques in our community [44, 45, 52] have been prototyped on USRPs and it is unclear if a low-power IoT endpoint device can decode concurrent transmissions in real-time within its stringent power and resource constraints. TinySDR enables us to explore such questions and design MAC protocols for decoding concurrent transmissions on IoT endpoints.

Using orthogonal LoRa codes. Here we explore a specific way of enabling concurrent transmissions in LoRa: using orthogonal codes. Specifically, to allow multiple LoRa nodes to communicate at the same time, we exploit LoRa's support for orthogonal transmissions [16] which can occupy the same frequency channel without interfering with each other. Two chirp symbols are orthogonal when they have a different chirp slope. For a chirp with a spreading factor of SF and bandwidth of BW, the chirp slope is given by: $\frac{BW^2}{2^{SF}}$ [47].

Decoding concurrent transmissions on tinySDR. In order to receive concurrent LoRa transmissions, tinySDR must be able to demodulate LoRa upchirp symbols with different slopes. Suppose we have two LoRa transmissions that use different spreading factor and bandwidth configurations: SF_1, BW_1 and SF_2, BW_2. To decode them concurrently, we implement decoders similar to Fig. 6b for each chirp configuration in parallel on our FPGA. Specifically, we first generate a corresponding downchirp symbol for each configuration in real-time using our chirp generator. Note that we generate each chirp with its corresponding configuration on the FPGA and we do not use pre-generated chirps on the FPGA. We then correlate the received signals with their corresponding downchirp symbols using time domain multiplication. After correlation, we take the appropriate length FFT of the result.

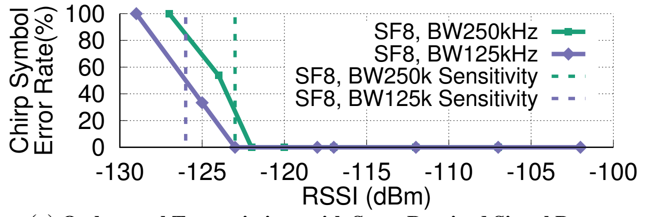
Evaluation. We evaluate three key aspects of our design: 1) the platform's effectiveness in decoding concurrent transmissions across a range of RSSI values, 2) the power consumption at the endpoint device while decoding concurrent transmissions and 3) the computational resources required.

We use two SX1276 LoRa transceivers as our transmitters and set them to transmit continuously at two different settings: they both use a spreading factor of $SF = 8$ but have two different bandwidth setups, $BW_1 = 125kHz$ and $BW_2 = 250kHz$. We set the two to send random chirp symbols. The tinySDR platform decodes these two concurrent transmissions and computes the chirp symbol error rate for each transmission. We evaluate two scenarios: 1) when the two transmitters have a similar power level at the receiver, 2) fix the power of one of the transmitters and increase the power of the other one.

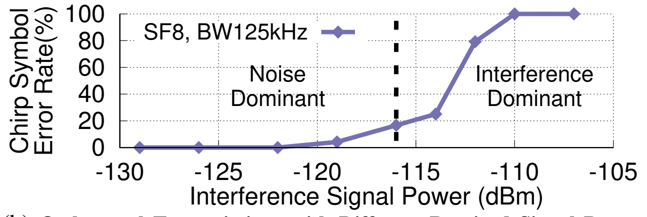
Fig. 15a shows the results when the two transmissions have similar power at the receiver. We lose around 2 dB and 0.5 dB sensitivity for concurrent demodulation of LoRa configurations with $BW_1 = 125kHz$ and $BW_2 = 250kHz$. This is because while in theory the two chirps are orthogonal, in practice, the chirps are created in the digital domain with discrete frequency steps which introduces some non-orthogonality.

Fig. 15b shows the results when the first LoRa transmitter $BW_1 = 125kHz$ is received near its sensitivity of -123 dBm and and the second LoRa transmitter changes its power. Here, the chirp symbol error rate is affected when the other transmission's power is higher than -116 dBm. When two concurrent transmissions are present, one acts as an interferer when decoding the other. The combined power of noise and the interferer, $P_{I,N}$, determines the error rate. When sweeping the power of interferer, at first the $P_{I,N}$ is dominated by noise and we should not see much effect on error rate. Then at some point their power would be equal which results in a 3 dB increase of $P_{I,N}$ and hence 3 dB sensitivity loss after which the error rate is determined by the interferer power. This demonstrates the need for power control for concurrent transmissions to be received on IoT endpoints.

Our parallel demodulation implementation, uses only 17% of the FPGA resources. This concurrent demodulation imple-



(a) **Orthogonal Transmissions with Same Received Signal Power.**



(b) **Orthogonal Transmissions with Different Received Signal Power**. We set the power of LoRa transmission with $BW_1 = 125kHz$ to -123 dBm and increase the power of the other one.

Figure 15: **Orthogonal LoRa Demodulation Evaluation**

mentation consumes 207 mW. Note that Semtech gateway solutions such as the SX1308 [29] can receive multiple transmissions. But, to the best of our knowledge we are the first to show that concurrent LoRa transmissions can be decoded on an IoT endpoint while meeting its power and computational requirements. This would have been difficult to do without tinySDR.

7 Conclusion and Research Opportunities

This paper presents the first SDR platform specifically tailored to the needs of IoT endpoints that can be used for large scale IoT network deployments. The goal of tinySDR is to provide a platform that can catalyze research in IoT networks.

Research on PHY/MAC protocols. TinySDR presents an opportunity for researchers to avoid the time consuming endeavor of building their own custom hardware and instead focus on PHY/MAC protocol innovations across the stack: What is the trade-off between packet length and overall throughput? Are there benefits of rate adaptation? What about concurrent transmissions from IoT devices? One could also create multi-hop IoT PHY/MAC innovations, which have not been explored well given the lack of a flexible platform.

Research on IoT localization. TinySDR could also be used to build localization systems as it gives access to I/Q signals and therefore phase across the 2.4 GHz and 900 MHz bands, which forms the basis for many localization algorithms [62]. One could also explore distributed localization solutions that combine the phase information across a distributed set of sensors to create a large MIMO sensing system.

Machine learning on IoT devices. The FPGA on tinySDR opens up exciting opportunities [42] for exploring machine learning algorithms on-board. This would allow researchers to explore trade-offs between the power overhead of running an on-board classifier versus sending data to the cloud. This could also enable use of high bandwidth sensors such as cameras and microphones where the power bottleneck may be communication rather than sensing.

Low power backscatter readers. Recent work on ambient backscatter [51, 53, 54, 59, 71] aims to achieve ultra-low power communication for IoT devices. Many of these proposals require either a single-tone generator [54] or a custom receiver to decode the backscatter transmissions [49, 50, 61, 65]. TinySDR can be used as a building block to achieve a battery-operated backscatter signal generation and receiver.

Better programming interface and protocols. In addition to IoT research opportunities, we can also improve our platform in multiple ways. TinySDR currently requires users to write Verilog or VHDL to program the FPGA and C code for programming the microcontroller. Future versions can incorporate a pipeline to use high level synthesis tools or integrate with GNUradio for easy prototyping. Further, tinySDR uses a simple MAC protocol for programming with a focus on using minimal system resources to allow for other custom software; however we could explore modified MAC protocols that simultaneously broadcast the updates across the network to reduce programming time.

Acknowledgments. We thank Aaron Schulman, Mohamad Katanbaf, and the anonymous reviewers for their helpful feedback on the paper. This work was funded in part by NSF awards CNS-1812554, CNS-1452494, CNS-1823148 and Google Faculty Research Awards.

References

[1] bladerf 2.0 micro. https://www.nuand.com/bladeRF_2_micro-brief.pdf.

[2] Limesdr. https://myriadrf.org/projects/limesdr/.

[3] Limesdr-mini. https://wiki.myriadrf.org/LimeSDR-Mini.

[4] An overview of lvds technology. http://www.ti.com/lit/an/snla165/snla165.pdf.

[5] Rtlsdr rtl2832u dvb-t tuner dongles. https://www.rtl-sdr.com/buy-rtl-sdr-dvb-t-dongles/.

[6] Usrp b200mini. https://www.ettus.com/content/files/USRP_B200mini_Data_Sheet.pdf.

[7] Usrp e310. https://www.ettus.com/content/files/USRP_E310_Datasheet.pdf.

[8] 2.45 ghz balun, filter combination, 2003. https://www.mouser.com/datasheet/2/611/JTI_Balun-Filter-2450FB15A050_2006-09-242325.pdf.

[9] Max5189 datasheet, 2003. https://datasheets.maximintegrated.com/en/ds/MAX5186-MAX5189.pdf.

[10] Max2831 datasheet, 2011. https://datasheets.maximintegrated.com/en/ds/MAX2831-MAX2832.pdf.

[11] Mdo4000b series datasheet, 2013. http://www.testequipmenthq.com/datasheets/TEKTRONIX-MDO4104B-6-Datasheet.pdf.

[12] Ad9364 transceiver datasheet, 2014. https://www.analog.com/media/en/technical-documentation/data-sheets/AD9364.pdf.

[13] Li-polymer battery technology specification, 2014. https://cdn-shop.adafruit.com/product-files/258/C101-_Li-Polymer_503562_1200mAh_3.7V_with_PCM_APPROVED_8.18.pdf.

[14] Smartsense temp/humidity manual, 2014. https://support.smartthings.com/hc/en-us/articles/203040294-SmartSense-Temperature-Humidity-Sensor.

[15] 3.5mhz, 500ma synchronous step down dc-dc regulator, 2015. https://www.semtech.com/uploads/documents/sc195.pdf.

[16] Lora modulation basics, 2015. https://www.semtech.com/uploads/documents/an1200.22.pdf.

[17] Ad9361 transceiver datasheet, 2016. https://www.analog.com/media/en/technical-documentation/data-sheets/AD9361.pdf.

[18] Ad9363 transceiver datasheet, 2016. https://www.analog.com/media/en/technical-documentation/data-sheets/AD9363.pdf.

[19] Adg904 datasheet by analog devices, 2016. http://www.analog.com/media/en/technical-documentation/data-sheets/ADG904.pdf.

[20] At86rf215 datasheet, 2016.

[21] Cc2650 simplelink datasheet by ti, 2016. http://www.ti.com/lit/ds/symlink/cc2650.pdf.

[22] Pcbminions inc., 2016. https://pcbminions.com/.

[23] Se2435l power amplifier datasheet, 2016. http://www.skyworksinc.com/uploads/documents/SE2435L_202412I.pdf.

[24] Sx1276 datasheet by semtech, 2016. https://www.semtech.com/uploads/documents/sx1276.pdf.

[25] Lms7002m datasheet, 2017. https://limemicro.com/app/uploads/2017/07/LMS7002M-Data-Sheet-v3.1r00.pdf.

[26] Lora alliance, 2017. https://lora-alliance.org/sites/default/files/2018-04/lorawantm_specification_-v1.1.pdf.

[27] Msp432p401r, msp432p401m simplelink mixed-signal microcontrollers datasheet, 2017. http://www.ti.com/lit/ds/symlink/msp432p401r.pdf.

[28] Sky66112 power amplifier datasheet, 2017. http://www.skyworksinc.com/uploads/documents/SKY66112_11_203225L.pdf.

[29] Sx1308 datasheet by semtech, 2017. https://www.semtech.com/uploads/documents/sx1308.pdf.

[30] Ad9228 datasheet, 2018. https://www.analog.com/media/en/technical-documentation/data-sheets/ad9228.pdf.

[31] Adalm-pluto overview, 2018. https://wiki.analog.com/university/tools/pluto.

[32] Atmel at86rf215 868/915/928 mhz impedance matched balun + lpf, 2018. https://www.mouser.com/datasheet/2/611/0896BM15E0025-1518705.pdf.

[33] Lfe5u fpga family datasheet, 2018. http://www.latticesemi.com/view_document?document_id=50461.

[34] minilzo implementation, 2018. http://www.oberhumer.com/opensource/lzo/#abstract.

[35] Sx1257 datasheet by semtech, 2018. https://www.semtech.com/uploads/documents/DS_SX1257_V1.2.pdf.

[36] Usrp x-300, 2018. https://www.ettus.com/product/details/X300-KIT.

[37] Bluetooth core specification v5.1, Jan. 2019. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[38] The things network, 2019. https://www.thethingsnetwork.org/.

[39] The things network arduino library, 2019. https://github.com/TheThingsNetwork/arduino-device-lib.

[40] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. R. Cavallaro, and A. Sabharwal. Warp, a unified wireless network testbed for education and research. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, pages 53–54. IEEE, 2007.

[41] N. Anand, E. Aryafar, and E. W. Knightly. Warplab: a flexible framework for rapid physical layer design. In *Proceedings of the 2010 ACM workshop on Wireless of the students, by the students, for the students*, pages 53–56. ACM, 2010.

[42] J. Chan, A. Wang, A. Krishnamurthy, and S. Gollakota. Deepsense: Enabling carrier sense in low-power wide area networks using deep learning. *CoRR*, abs/1904.10607, 2019.

[43] P. Dutta, Y.-S. Kuo, A. Ledeczi, T. Schmid, and P. Volgyesi. Putting the software radio on a low-calorie diet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 20. ACM, 2010.

[44] R. Eletreby, D. Zhang, S. Kumar, and O. Yağan. Empowering low-power wide area networks in urban settings. SIGCOMM '17.

[45] S. Gollakota and D. Katabi. Zigzag decoding: Combating hidden terminals in wireless networks. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, 2008.

[46] Y. Guddeti, R. Subbaraman, M. Khazraee, A. Schulman, and D. Bharadia. Sweepsense: Sensing 5 ghz in 5 milliseconds with low-cost radios. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 317–330, 2019.

[47] M. Hessar, A. Najafi, and S. Gollakota. Netscatter: Enabling large-scale backscatter networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 271–284, 2019.

[48] R. Hildebrandt. The pseudo dual-edge d-flip-flop, 2011.

[49] V. Iyer, J. Chan, I. Culhane, J. Mankoff, and S. Gollakota. Wireless analytics for 3d printed objects. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 141–152, New York, NY, USA, 2018. ACM.

[50] V. Iyer, J. Chan, and S. Gollakota. 3d printing wireless connected objects. *ACM Trans. Graph.*, 36(6), Nov. 2017.

[51] V. Iyer, V. Talla, B. Kellogg, S. Gollakota, and J. Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of the 2016 ACM SIGCOMM Conference*.

[52] S. Katti, S. Gollakota, and D. Katabi. Embracing wireless interference: Analog network coding. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 397–408. ACM, 2007.

[53] B. Kellogg, A. Parks, S. Gollakota, J. R. Smith, and D. Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*.

[54] B. Kellogg, V. Talla, S. Gollakota, and J. R. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *NSDI 16*.

[55] A. Khattab, J. Camp, C. Hunter, P. Murphy, A. Sabharwal, and E. W. Knightly. Warp: a flexible platform for clean-slate wireless medium access protocol design. *ACM SIGMOBILE Mobile Computing and Communications Review*, 12(1):56–58, 2008.

[56] Y.-S. Kuo, P. Pannuto, T. Schmid, and P. Dutta. Reconfiguring the software radio to improve power, price, and portability. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 267–280. ACM, 2012.

[57] Y.-S. Kuo, T. Schmid, and P. Dutta. A compact, inexpensive, and battery-powered software-defined radio platform. In *Proceedings of the 11th international conference on Information Processing in Sensor Networks*, pages 137–138. ACM, 2012.

[58] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Soda: A low-power architecture for software radio. *ACM SIGARCH Computer Architecture News*, 34(2):89–101, 2006.

[59] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith. Ambient backscatter: Wireless communication out of thin air. SIGCOMM '13.

[60] G. J. Minden, J. B. Evans, L. Searl, D. DePardo, V. R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, et al. Kuar: A flexible software-defined radio development platform. In *2007 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks*, pages 428–439. IEEE, 2007.

[61] S. Naderiparizi, M. Hessar, V. Talla, S. Gollakota, and J. R. Smith. Towards battery-free hd video streaming. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[62] R. Nandakumar, V. Iyer, and S. Gollakota. 3d localization for sub-centimeter sized devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, 2018.

[63] R. Narayanan and S. Kumar. Revisiting software defined radios in the iot era. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. ACM, 2018.

[64] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, H. Balakrishnan, et al. Airblue: A system for cross-layer wireless protocol development. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, page 4. ACM, 2010.

[65] A. Saffari, M. Hessar, S. Naderiparizi, and J. R. Smith. Battery-free wireless video streaming camera system. In *2019 IEEE International Conference on RFID (RFID)*, pages 1–8. IEEE, 2019.

[66] C. Shepard, A. Javed, and L. Zhong. Control channel design for many-antenna mu-mimo. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15.

[67] C. Shepard, H. Yu, N. Anand, E. Li, T. Marzetta, R. Yang, and L. Zhong. Argos: Practical many-antenna base stations. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12.

[68] P. D. Sutton, J. Lotze, H. Lahlou, S. A. Fahmy, K. E. Nolan, B. Ozgul, T. W. Rondeau, J. Noguera, and L. E. Doyle. Iris: an architecture for cognitive radio networking testbeds. *IEEE communications magazine*, 48(9):114–122, 2010.

[69] V. Talla, M. Hessar, B. Kellogg, A. Najafi, J. R. Smith, and S. Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2017.

[70] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker. Sora: high-performance software radio using general-purpose multi-core processors. In *6th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 09)*, 2009.

[71] A. Wang, V. Iyer, V. Talla, J. R. Smith, and S. Gollakota. FM backscatter: Enabling connected cities and smart fabrics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[72] H. Wu, T. Wang, Z. Yuan, C. Peng, Z. Li, Z. Tan, B. Ding, X. Li, Y. Li, J. Liu, et al. The tick programmable low-latency sdr system. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 101–113. ACM, 2017.

[73] J. Zhang, X. Zhang, P. Kulkarni, and P. Ramanathan. Openmili: a 60 ghz software radio platform with a reconfigurable phased-array antenna. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 162–175. ACM, 2016.

RFocus: Beamforming Using Thousands of Passive Antennas

Venkat Arun and Hari Balakrishnan
MIT, CSAIL
{venkatar, hari}@csail.mit.edu

Abstract

To reduce transmit power, increase throughput, and improve range, radio systems benefit from the ability to direct more of the transmitted power toward the intended receiver. Many modern systems beamform with antenna arrays for this purpose. However, a radio's ability to direct its signal is fundamentally limited by its size. This limitation is acute on IoT and mobile devices, which are small and inexpensive, but even access points and base stations are typically constrained to a modest number of antennas.

To address this problem, we introduce *RFocus*, which moves beamforming functions from the radio endpoints to the environment. RFocus includes a two-dimensional surface with a rectangular array of simple RF switch elements. Each switch element either lets the signal through or reflects it. The surface does not emit any power of its own. The state of the elements is set by a software controller to maximize the signal strength at a receiver, with a novel optimization algorithm that uses signal strength measurements from the receiver. The RFocus surface can be manufactured as an inexpensive thin "wallpaper". In one floor of an office building (at MIT CSAIL), our prototype improves the median signal strength by $9.5\times$ and the median channel capacity by $2.0\times$.

1 Introduction

Many radio systems use directional or sectorized antennas and beamforming to improve the throughput or range of a wireless communication link. Beamforming ensures that a larger fraction of transmitted energy reaches the intended receiver,while reducing unintended interference. A radio with many antennas spread densely over a large area can fundamentally beamform better than a smaller radio [1, 4]. However, there are many practical challenges to making radio systems with large antenna arrays. First devices such as IoT sensors and handhelds must be small in size. Second, connecting each antenna in an array to full-fledged radio transmit/receive circuitry increases cost and power. Third, large, bulky systems are hard to deploy, even in infrastructure base stations or access points.

To address these challenges and achieve the equivalent of a large number of antennas, we propose *RFocus*. RFocus includes a *software-controlled surface* placed in the environment, made up of thousands of simple switching elements. RFocus also has a *controller* that configures each element so that the surface, as a whole, directs a signal from a transmitter to a receiver without any signal amplification (i.e., no extra power) or knowledge of the locations of the transmitter or receiver. This approach moves the task of beamforming from the transmitter to the surface. Any device in the vicinity of the surface can reap the benefits of RFocus without itself being large or consuming additional energy.

The RFocus surface is made up of thousands of simple elements organized in a rectangular array. To reduce cost and energy, each element is a single simple 2-way RF switch. Each element in the RFocus surface can be in one of two states. When "on", the element reflects the signal; otherwise, the signal passes through. Each receiver periodically sends measurements of the received signal strength to the RFocus controller. The controller uses these to configure the elements on the RFocus surface to maximize the received signal strength. This optimization has two steps: a training phase, where the controller configures test states on the surface and monitors the changes in the reported measurements to collect data, and an optimized phase where the controller uses this data to set the on/off state for each element. RFocus can switch between optimized states to maximize signal strength between different pairs of endpoints.

RFocus can work both as a "mirror" or a "lens", with the controller choosing the right mode. That is, radio endpoints can be on the same side of the surface, or on opposite sides.

We have built an RFocus prototype with 3,200 inexpensive[1] antennas on a 6 square-meter surface. This configuration may well be the largest published number of antennas ever used for a single communication link.

The controller's optimization algorithm solves three key challenges. First, indoor environments exhibit complex multi-path, and the direct path may be blocked. Therefore the optimal configuration might not correspond to directing the signal along a single direction, but can depend on the environment. Second, the effect of an individual element on the channel is usually miniscule, and hence hard to measure (§6.2.3). Third, the phase of a channel is hard to

[1] At scale, the cost of each element is a few cents.

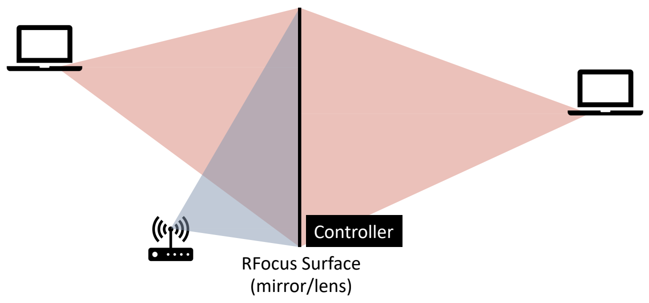


Figure 1: RFocus is an electronically configurable mirror/lens that focuses the transmitter's signal on the receiver.

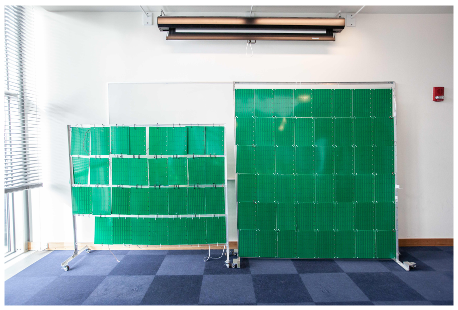


Figure 2: Our prototype of the RFocus surface.

measure when the signal is weak, due to carrier frequency offset and radio-clock jitter. And last but not least, commodity radio receivers usually don't report phase.

The contributions of this paper include:

1. The RFocus controller, incorporating two key ideas. First, it modulates all elements at once to boost the effect of the RFocus surface on the channel, hence making the change large enough to be measurable. Second, it relies only on signal strength measurements, sidestepping difficulties in measuring phase. We provide an analysis of the algorithm under some assumptions, that suggests that it finds a solution that is close to optimal.

2. Experiments, which show that in a typical indoor office environment, RFocus achieves a median $9.5\times$ improvement in signal strength and $2.0\times$ improvement in channel capacity. Moreover, RFocus is robust to element failure; even when one-third the elements fail, the relative performance improvement does not plummet to 0, but drops by 50%.

2 Related Work

Passive reflective surfaces to control incident radiation, called "meta-surfaces" have been studied extensively by applied physicists in the microwave, terahertz, and optical frequencies [2, 6–8, 13]. These works focus on hardware design to control the incident radiation efficiently. Some of these metasurfaces are configurable. Here, given the incident wave and desired reflected/transmitted wave direction(s), they develop algorithms to determine good configurations, for example to create programmable holograms. Recently there has been interest in using metasurfaces, called "reconfigurable intelligent surfaces", to improve communication channels. Most papers explore the idea with theory and simulation [12, 17–19, 28] except for a few papers providing small-scale experimental results [11, 15, 29] that evaluate channel improvement between one fixed pair of endpoints that are close to each other and to the surface.

In prior work, the method to determine a good configuration works by varying the state of the elements *one at a time* and measuring its effect on the channel. This approach is linear in the number of elements, but requires the receiver to be able to accurately measure the impact of the change in a single element. If the transmitter and receiver are very close to the surface, the effect is measurable, but not otherwise because the strength of the signal attenuates as at least $1/(d_s d_r)^2$, where d_s and d_r are the distances from the element to the sender and receiver, respectively. For a large surface with thousands of antennas, the effect becomes hard to measure because some elements will necessarily be at long distances from the transmitter or receiver §6.2.3. The reason is that the size of each element is between one-quarter to one-half of the wavelength of the signal (at 2.4 GHz, the wavelength is 15 cm), so thousands of elements require a surface area of a few square meters, and at a distance of a few meters, measuring a single-element change is inaccurate especially on commodity radio receivers. Our algorithm overcomes this problem and scales to larger surfaces by varying many elements at once and boosting the signal strength changes being measured.

LAIA [16, 32] is a recent project that also achieves gains at larger distances, but with a different setup and design from RFocus. LAIA helps endpoints whose line-of-sight path is blocked by a wall. LAIA elements collect radio energy from one side of the wall, phase-shift them, and take them to the other side of the wall via wires traversing holes in the wall. Because they go through the wall with wires, individual LAIA elements have a much larger impact on the channel than RFocus elements, allowing LAIA to function with fewer elements than RFocus, and without RFocus's optimization algorithm.

Another line of related work improves wireless coverage by analyzing the indoor space, and custom-designing a 3D reflector for that space. When 3D-printed and placed behind an access point (AP), the reflector reflects energy in specific directions to maximize signal strength at previously uncovered areas [5, 33]. Once deployed, this reflector has a low operational cost because it is just a passive metal-coated object. But a new reflector needs to be designed for each new location and whenever the space changes or the AP is moved or a new AP is added. Moreover, a single solution has to be designed for all pairs of endpoints, whereas RFocus dynamically and automatically configures a new pattern for each pair.

RFocus can be thought of as a phased array [25, 26] that uses the air instead of transmission lines to distribute signals from the transmitter to the antennas. We believe this allows larger antenna arrays to be deployed more ubiquitously. Transmission lines tend to be expensive and bulky. Even PCB-trace transmission lines, which are easy to manufacture, depend on properties of the dielectric and cannot be manufactured as a paper-thin sheet. This is unlike the wires used in RFocus, which are digital and do not carry high-speed signals. Due to its simplicity, RFocus can be readily incorporated into the environment (§7). On the other hand, for the same number of antennas, RFocus achieves a lower gain than phased arrays because it doesn't harness all of the transmitted energy (§7).

Digital Light Processors (DLPs) and Digital Micromirror Devices (DMDs) [22, 23, 27] are micro-electro-mechanical systems (MEMS) devices that have a large array of small (microns or smaller) mirrors on a single silicon chip. The mirrors can be toggled between two angles, controlling which ones reflect light onto the object, and which reflect it elsewhere. They are used for steering lasers and in display projectors. RFocus could be thought of as a DMD for radio.

Range extenders increase signal strength at the receiver. However, by rebroadcasting signals, they increase interference and reduce transmission opportunities. By precisely focusing energy already available, RFocus decreases interference while increasing signal strength. That said, RFocus complements range extenders and both could be used together if necessary.

Reconfigurable antennas [10] and reflectarray antennas [14] have RF switches and phase shifters, which allow them to dynamically change their characteristics such as operating frequency, input impedance, and directionality. These approaches modify an antenna to improve characteristics. By contrast, RFocus leaves the transmit and receive antennas unmodified, instead modifying the environment to improve communication for all nearby devices. Additionally, like phased arrays, it is difficult to use wires to scale to a large number of radiating elements.

3 Background

3.1 System Model and Notation

In most environments, there are multiple paths between a sending and receiving antenna. For a narrow-band signal, the effect of each path can be represented by a complex number. The net effect of the channel is the sum of the effects of all the paths. A subset of the paths pass via each of the N elements on the RFocus surface. Denote the contribution of the elements by $c_1, \ldots, c_N$. We combine all the paths *not* going via RFocus into one complex number c_E (E for environment). The net channel, h, is equal to $c_E + \sum_{i=1}^{N} \alpha_i c_i$, where α_i represents the amplitude change and phase shift introduced by element i.

RFocus controls the channel, h, by adjusting α_i. c_E and c_i are functions of the path lengths. α_i depends on three factors: the state of the i^{th} element and its neighbors; the shape of the antennas composing the element, and the angles at which the path enters and leaves the i^{th} element. Since the elements are passive and lack power, $|\alpha_i| \leq 1$.

Maximizing $|h|$ maximizes the received signal strength. The maximum possible value of $|h|$, the sum of several complex numbers, is the sum of the magnitudes of the summands:

$$|h_{\max}| = |c_E| + \sum_{i=1}^{N} |c_i| \tag{1}$$

This maximum is achievable if we had full control over each α_i by setting $\alpha_i = \frac{c_E}{|c_E|}\frac{c_i^*}{|c_i|}$, where c_i^* denotes conjugation. Unfortunately, we do not have full control over α_i. Each element can only be on or off. If we assume that α_i is a function of only the i^{th} element's state and not its neighbors (we validate this assumption with experiments in §6.2.1), then we can write the channel as

$$h = h_Z + \mathbf{h} \cdot \mathbf{b} = h_Z + \sum_{i=1}^{N} \mathbf{h}^{(i)} \mathbf{b}^{(i)} \tag{2}$$

Here $\mathbf{b}^{(i)} \in \{0, 1\}$ denotes whether the i^{th} element is off or on; h_Z is the channel when all elements are off; and $\mathbf{h}^{(i)}$ is the effect of turning the i^{th} element on. Here, we have folded the complexities of α_i into c_i to get $\mathbf{h}^{(i)}$. We in prove in §5.2 that having the ability to set *any* $|\mathbf{b}^{(i)}| \leq 1$ (i.e., fine-grained control over the phase) instead of only 0 or 1 gives only a factor-of-π advantage in optimizing $|h|$.

3.2 How Size Helps Communication

To get a qualitative understanding of the benefits of RFocus, we use some well known results from physics. First, RFocus can only control the portion of transmitted energy that falls on it. This is given by the solid angle, Ω, subtended by the surface on the transmitter. The Abbe diffraction limit tells us how well the surface can focus the energy at the receiver. If the surface modulates the incident energy perfectly, the spot onto which the energy is focused will have an area a proportional to $\lambda^2\left(1 + 4\frac{d^2}{A}\right)$, where A is the area of the RFocus surface and d is the distance of the receiver from it [20]. Thus a fraction $\Omega/(4\pi)$ of the transmitted energy is now spread over an area $\approx a$, and hence peak energy will be proportional to Ω/a. RFocus works best when either the transmitter or the receiver is close to the surface, and therefore Ω is large (as a rule of thumb, about 3 meters for a surface 2-3 meters in length/width). Note, the Abbe diffraction limit is only an approximate result.

A/d^2 is proportional to the angle subtended by the surface on the receiver. There are two regimes depending on how large this angle is. If the radio is far away from the surface (i.e. $d^2 \gg A$), then $a \underset{\sim}{\propto} d^2/A$ and energy falls as $A\Omega/d^2$. This is still a $1/d^2$ fall, but the constant is improved by a factor of $A\Omega$. If

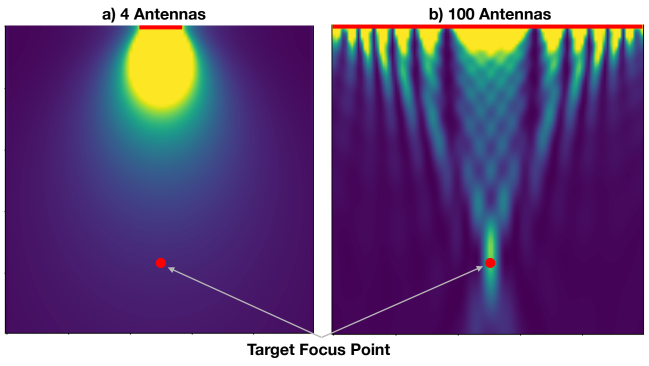


Figure 3: A simulation of how signal strength (brighter is higher) is distributed when an antenna array tries to maximize signal strength at the target point. The antennas are on the top horizontal line (red). With four antennas the signal quickly begins to diffuse. The 100 antennas subtend a large angle at the target, and are hence able to focus energy there, avoiding attenuation due to spreading.

the radio is closer, then $d^2 \sim A$, and the first term dominates. In this regime, a can be made quite small, on the order of a few λ^2. Hence almost all of the transmitted energy can be incident on the receiver.

In traditional beamforming, A is typically small, hence $d^2 \gg A$ and we are always in the first regime where the signal experiences a $1/d^2$ attenuation as it spreads out. The difference between the two regimes is illustrated in Figure 3.

4 Optimization Algorithm

The RFocus controller uses measurements from the radio endpoints to maximize signal strength at the receiver. In this section we first describe the challenges in measuring changes in the channel, and why we rely only on RSSI measurements (§4.1). Then we describe our algorithm (§4.2) before giving a preliminary theoretical analysis showing that it converges to a near-optimal solution (§4.3).

4.1 Measuring the Channel

A direct, but naive, method: When all the elements are turned off, the channel is h_Z, by definition. Ideally, to measure each $\mathbf{h}^{(i)}$, we could turn just the i^{th} element on, and measure the difference from h_Z. But this change is usually too feeble to measure because an element is just a small piece of metal lying somewhere in the environment (§6.2.3). In many cases, we cannot statistically tell the difference between an element being on or off, even with hundreds of samples. This is also the reason why gradient descent to find an optimal configuration is untenable.

Boosting the signal. Each $\mathbf{h}^{(i)}$ may be small, but all the elements together can have a large effect. To make the change measurable, we generate several configurations of the surface by randomly choosing the state of each element. If we vary N elements, the change in the channel will have an expected magnitude of $O(\bar{h}\sqrt{N})$ (due to the central limit theorem), where $\bar{h}$ is the average magnitude of $\mathbf{h}^{(i)}$. This gives us a $O(\sqrt{N})$ boost in amplitude (and $O(N)$ boost in signal strength) over flipping just one element.

Challenges in measuring phase. One might consider a method that would measure the (complex) channel for many random configurations of the surface, and solve the linear equations to obtain all the $\mathbf{h}^{(i)}$. But this is difficult because it needs to measure changes in the *phase* of the channel, which commodity wireless devices don't provide. More importantly, the change in the channel, even after this $O(\sqrt{N})$ boost, remains small. Measuring small changes in phase is hard because the clocks of the transmitter and receiver are never perfectly synchronized, leading to carrier frequency offset (CFO) and jitter in phase measurements. Correcting for CFO over long periods of time is non-trivial, since CFO drifts over time.

Using RSSI. Our method uses only signal-strength measurements, ignoring phase information altogether. RSSI is not always an absolute metric, and may vary due to automatic-gain control changes or temperature changes at the amplifier. Hence we measure RSSI of a test state *relative* to a baseline state; e.g., to the all-zeros state where all elements are turned off. We call this quantity the *RSSI-ratio*.

4.2 RFocus's Optimization Algorithm

Algorithm 1 The majority voting algorithm (described below)

```
procedure MajorityVote([$\mathbf{b}_1,\ldots,\mathbf{b}_K$], RSSI)
    // RSSI[$j$] gives the measured RSSI-ratio for state $\mathbf{b}_j$
    $\mathbf{b}_\perp \leftarrow$ blank-vector            ▷ The final optimized state
    $m \leftarrow$ Median(RSSI)
    for $i := 0$ to $N$ do
        VoteOn $\leftarrow 0$, VoteOff $\leftarrow 0$
        for $j := 0$ to $K$ do
            if
                ($\mathbf{b}_j^{(i)} == 1$ and RSSI[$j$] $> m$) or
                ($\mathbf{b}_j^{(i)} == 0$ and RSSI[$j$] $< m$) then
                VoteOn $\leftarrow$ VoteOn $+ 1$
            else
                VoteOff $\leftarrow$ VoteOff $+ 1$
        $\mathbf{b}_\perp^{(i)} \leftarrow$ (VoteOn $>$ VoteOff)
    return $\mathbf{b}_\perp$
```

Given the measured RSSI-ratio for a set of K random configurations, the idea is to compare the RSSI-ratio of each measurement to the median value: if the RSSI-ratio when the i^{th} element is on (off) is higher than the median, then we add a vote for the element to be on (off) in the optimized configuration. The i^{th} element's optimized state is the state that received more votes for that element. Algorithm 1 gives the pseudo-code for how the controller determines the optimized state for each bit, i.

We use the median instead of the mean of the RSSI-ratio samples because it is more robust to one-shot noise, which may occur due to people moving or amplifier changes. We expect this algorithm to be robust to element failure and occluded elements, since elements' states are determined independently of each other. We evaluate how the received signal strength of the optimized configuration degrades with element failures in Section 6.4.

4.3 Theoretical Analysis

We show that if each $\mathbf{h}^{(i)}$ is small and the collection has uniformly-distributed phases, the optimization algorithm will find a near-optimal solution with high probability.

4.3.1 Assumptions

The proof relies on the following assumptions:

Assumption 1. *The phases of* $\mathbf{h}^{(i)}$ *are uniformly distributed random variables in* $(-\pi,\pi]$.

This is reasonable because the paths to different elements are of different lengths, so all phases are equally likely to be represented. To remove any spatial correlations, we can randomize the indices of $\mathbf{h}^{(i)}$.

Assumption 2. *For uniformly random* $\mathbf{b}$, $|\mathbf{b}\cdot\mathbf{h}| \ll h_Z$ *with high probability.*

The average of $\mathbf{h}^{(i)}$ is 0 when phases are uniformly random. Hence, assumption 1 implies that for uniformly random $\mathbf{b}$, $|\mathbf{h}\cdot\mathbf{b}|$ is $O(\frac{1}{\sqrt{N}}\sum_{i=1}^{N}\left|\mathbf{h}^{(i)}\right|)$ due to the central limit theorem.

Though $|\mathbf{h}\cdot\mathbf{b}|$ is small for random $\mathbf{b}$, an *optimal* assignment to $\mathbf{b}$ can cause $|\mathbf{h}\cdot\mathbf{b}|$ be large relative to $|h_Z|$, since it will be $O(N)$ times bigger than the average $|\mathbf{h}^{(i)}|$ (see theorem 2). On the other hand, for random $|\mathbf{b}|$, $|\mathbf{h}\cdot\mathbf{b}|$ is only $O(\sqrt{N})$ times bigger than the average $|\mathbf{h}^{(i)}|$.

Assumption 3. $|\mathbf{h}^{(i)}|$ *is bounded above by a constant, even as* $N\to\infty$.

With this assumption, $\left|\mathbf{h}^{(i)}\right| \ll \sum_{j=i}^{N}\left|\mathbf{h}^{(j)}\right|$, capturing the intuition that, since all elements are small, there is no dominating element. Many elements must contribute to create a large effect.

Note that an RFocus element is only half a wavelength long when on. This implies that, individually, an element cannot be very directional [9]. Hence, the reflections are not specular. Rather, each element diffracts waves to a wide range of angles. Hence, we would expect all the $\mathbf{h}^{(i)}$ to have a similar (likely, small) magnitude. However, acting together, the elements form a large structure which which c which and can beconfigured to achieve directionality (§3.2). In addition, we assume that equation 2 is accurate, which we later verify experimentally (see §6.2.1). Our analysis is in the limit where the number of elements and the number of measurements go to infinity.

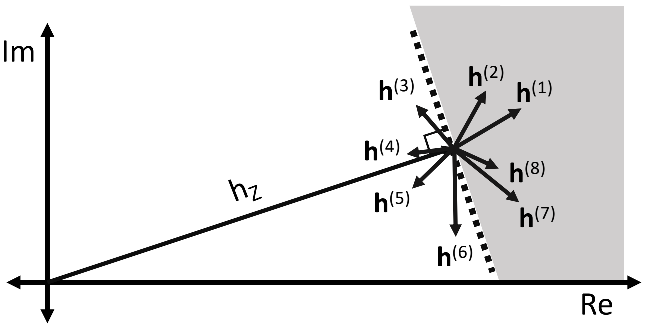


Figure 4: h_Z and the $\mathbf{h}^{(i)}$ in the complex plane. To maximize signal strength, we should pick the $\mathbf{h}^{(i)}$ in the shaded region and remove the others.

4.3.2 Proof Outline

Consider h_Z and $\mathbf{h}^{(i)}$ on the complex plane as shown in Figure 4. The signal strength is $|h_Z+\sum_i \mathbf{h}^{(i)}\mathbf{b}^{(i)}|^2$. If all the elements are turned on, the $\mathbf{h}^{(i)}$ interfere destructively with each other. To increase signal strength, we should eliminate this destructive interference. To do so, we should pick a half-plane (the dotted line), turn on all elements on one side of it and turn the others off. We show in Lemma 1 that the optimal solution takes this form.

Next, if the phases of $\mathbf{h}^{(i)}$ are randomly distributed, it doesn't matter which half-plane we pick; they will all produce roughly the same $|\mathbf{b}\cdot\mathbf{h}|$ (Lemma 2). Hence, we can maximize alignment with h_Z by picking the half-plane perpendicular to it (pictured in the figure).

The controller needs to determine for each element i if $\mathbf{h}^{(i)}$ is aligned with h_Z. Or equivalently, whether $\Re(\mathbf{h}^{(i)}h_Z^*) > 0$ where $\Re(\cdot)$ denotes the real component of a complex number and * denotes complex conjugation. The algorithm takes advantage of the fact that if $\Re(\mathbf{h}^{(i)}h_Z^*) > 0$, the signal strength is slightly more likely to be higher than the median signal strength when $\mathbf{h}^{(i)}$ is turned on (Theorem 1).

4.3.3 A Near-Optimal Solution

We show that all the chosen $\mathbf{h}^{(i)}$ in an optimal solution lie on one side of a line in the complex plane, formalized as follows:

Lemma 1. *Under assumptions 2 and 3, let* $\mathbf{b}_{OPT}$ *be an optimal state assignment. Then,* $\mathbf{b}_{OPT}^{(i)} = 1$ *if and only if* $\Re(\mathbf{h}^{(i)}\cdot H(\mathbf{b}_{OPT})^*) > 0$, *where* $H(\mathbf{b}) = h_Z + \mathbf{h}\cdot\mathbf{b}$.

Proof. If this were not the case, we could flip $\mathbf{b}_{OPT}^{(i)}$ to get

$$\begin{aligned}
&|H(\mathbf{b}_{OPT})-\mathbf{b}_{OPT}^{(i)}\mathbf{h}^{(i)}+\left(1-\mathbf{b}_{OPT}^{(i)}\right)\mathbf{h}^{(i)}| \\
&=\left|H(\mathbf{b}_{OPT})+\left(1-2\mathbf{b}_{OPT}^{(i)}\right)\mathbf{h}^{(i)}\right| \qquad (3)\\
&\geq |H(\mathbf{b}_{OPT})|
\end{aligned}$$

To prove the last inequality, note

$$\begin{aligned}
&\left|H(\mathbf{b}_{OPT})+\left(1-2\mathbf{b}^{(i)}\right)\mathbf{h}^{(i)}\right|^2-|H(\mathbf{b}_{OPT})|^2 \\
&=\left|\left(1-2\mathbf{b}^{(i)}\right)\mathbf{h}^{(i)}\right|^2+2\Re\left(\left(1-2\mathbf{b}^{(i)}\right)\mathbf{h}^{(i)}\cdot H(\mathbf{b}_{OPT})^*\right) \qquad (4)\\
&\approx 2\Re((1-2\mathbf{b}^{(i)})\mathbf{h}^{(i)}\cdot H(\mathbf{b}_{OPT})^*)\\
&\geq 0
\end{aligned}$$

The second-last approximation follows because $|\mathbf{h}^{(i)}|$ is bounded and hence $|\mathbf{h}^{(i)}| \ll h_Z \leq |H(\mathbf{b}_{OPT})|$. The last inequality holds if our condition on $\mathbf{b}_{OPT}^{(i)}$ is not satisfied. If so, the solution can be improved, which contradicts our assumption that $\mathbf{b}_{OPT}$ is optimal. □

The next lemma states that the solution pictured in Figure 4 is close to optimal with high probability as the number of elements $N \to \infty$.

Lemma 2. *Let $\mathbf{b}_{OPT}$ be the optimal assignment that maximizes $|h_Z+\mathbf{h}\cdot\mathbf{b}|$ and $\mathbf{b}_\perp$ be such that the i^{th} component $\mathbf{b}_\perp^{(i)}=1$ if and only if $\Re(\mathbf{h}^{(i)}\cdot h_Z^*)>0$. As $N\to\infty$, if assumptions 1 and 3 hold, then $|H(\mathbf{b}_{OPT})|/|H(\mathbf{b}_\perp)|<1+\varepsilon\ \forall\varepsilon>0$, with high probability.*

The proof is in Appendix A. We give the outline here. Let $\mathbf{b}_\theta$ be the state such that $\mathbf{b}_\theta^{(i)}=1$ iff $\Re(\mathbf{h}^{(i)}e^{-j\theta})>0$. We show that, since the phases of $\mathbf{h}$ are uniformly distributed, it does not matter what θ we pick. That is, $\max_\theta|\mathbf{b}_\theta\cdot\mathbf{h}|$ isn't very different from $\min_\theta|\mathbf{b}_\theta\cdot\mathbf{h}|$. Hence, we can pick θ to be the phase of h_Z to prove the theorem.

4.3.4 Analysis of the Algorithm

We prove that Algorithm 1 finds the near-optimal solution discussed in Lemma 2.

Theorem 1. *Let assumptions 1, 2, and 3 hold. Then, as $N\to\infty$ and $K\to\infty$, the configuration returned by Algorithm 1 finds a near-optimal solution.*

Proof. According to Equation 2, when we randomly vary $\mathbf{b}^{(i)}$, h becomes a random variable, H, with mean $h_Z+S/2$, where $S=\sum_{i=1}^N\mathbf{h}^{(i)}$ is the sum of the contributions of the rest of the elements. The $S/2$ term appears because we include each element with probability $1/2$. Figure 5(a) shows this probability distribution. Consider an element i that hasn't yet been fixed. If we condition the probability distribution on the $i$thelement being on, then the PDF shifts by $\mathbf{h}^{(i)}/2$ as

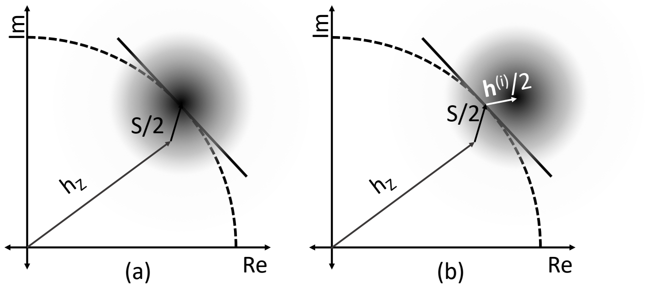


Figure 5: (a) Shows the probability density function of the channel when all bits are chosen uniformly at random. (b) Shows the PDF conditioned on the i^{th} bit being 'on'. The dashed circle is centered at the origin with radius $|h_Z+S/2|$, where $S=\sum_{i=1}^N\mathbf{h}^{(i)}$. Depending on which side of the circle h_i takes the mean, the mean magnitude will be greater or lesser than that of the unconditional PDF.

shown in Figure 5(b), because the $i$thelement's value is fixed in these samples (it shifts by $\mathbf{h}^{(i)}/2$ and not $\mathbf{h}^{(i)}$ since we have already included the other $\mathbf{h}^{(i)}/2$ in $S/2$). If we condition on the element being off, then the mean shifts by $-\mathbf{h}^{(i)}/2$.

The central limit theorem implies that $H/\sqrt{N}$ is Gaussian as $N\to\infty$. Hence, if $\mathbf{h}^{(i)}/2$ (or $-\mathbf{h}^{(i)}/2$) shifts the mean to be outside the circle, then when the element is on (or off) the RSSI-ratio is more likely to be greater than the unconditional mean. If $\mathbf{h}^{(i)}/2$ shifts the mean inside the circle, then the opposite holds. Here, we use assumption 2, which implies that H isn't shifted far from $h_Z+S/2$. Hence, as $K\to\infty$, Algorithm 1 determines with confidence tending to 100%, whether the conditional mean is inside or outside the circle, by looking at RSSI alone. Note that mean equals median in this case.

Assumption 2 implies that $|\mathbf{h}^{(i)}|\ll|h_Z|$. Thus, the mean shifts outside the circle if and only if it shifts to the outer side of the tangent line shown in figure 5. That is, $\mathbf{b}_\perp^{(i)}=1$ if and only if $\Re\left(\mathbf{h}^{(i)}\cdot(h_Z+S/2)^*\right)$. But $h_Z+S/2\approx h_Z$, because of assumption 2. Hence $\mathbf{b}_\perp^{(i)}=1$ if and only if $\Re\left(\mathbf{h}^{(i)}\cdot h_Z^*\right)>0$. From Lemma 2, we know that this produces a near-optimal solution. □

To a first order approximation, the RSSI-ratio is linear in $\mathbf{b}$.[2] Hence linear regression is an alternative to majority voting. Majority voting has three advantages. First, unlike linear regression, majority voting is robust to outliers. Second, it is conceptually and computationally simpler, which is particularly important when the controller is an embedded device. Third, when the votes for a particular element determine its value with high confidence (say, $>95\%$), it can be fixed for the rest of the training

[2] RSSI-ratio, $|h/h_Z|=|1+\mathbf{b}\cdot\mathbf{h}/h_Z|\approx 1+\sum_i\Re(\mathbf{b}^{(i)}\mathbf{h}^{(i)}/h_Z)$ because $|\mathbf{b}\cdot\mathbf{h}|\ll h_Z$ for random $\mathbf{b}$ and $|1+x|\approx 1+\Re(x)$ for small x.

period. This allows the channel to be improved even before the training ends. Fixing elements also removes the confounding effect they have on determining values for other elements. An earlier version of this paper used this idea [3]. However due to a performance optimization (§6.1), we do not implement it here.

5 Antenna Array Design

We have made two key design choices in the physical design of our antenna array. First, each element has only two states: one that reflects the signal, another that lets it through. Second, each element is half a wavelength tall and $1/10$ of a wavelength wide. In this section we explore the tradeoffs in these choices.

5.1 How Big Should Each Element Be?

An array with many small antennas gives better control over the reflected signal, while one with fewer but larger antennas is cheaper and simpler. If the inter-antenna spacing is larger than half a wavelength, there will be grating lobes where the signal is sent in directions other than the one desired [4].

Designing antennas that are much smaller than a wavelength[3] is challenging. Small antennas are either inefficient, absorbing only a small fraction of incident energy, or they are efficient only over a small bandwidth [21, 31]. Further, when placed close to each other, antennas interact strongly with each other in a way that is often hard to model.

Fortunately, a well understood result states that controlling the reflected wave at a granularity finer than half a wavelength gives only marginal benefits. Consider two infinite parallel planes a distance d apart, separated by a homogeneous medium. Variations in electric/magnetic fields in one plane that are faster than once per wavelength, will have a negligible effect on the fields on the other plane; it decays exponentially with d [24]. Hence any fine-grained variations we introduce in the surface will be lost as soon as the signal propagates a few wavelengths in either direction. Thus we can design an array with antennas comparable to a wavelength and still get most of the benefits. Metasurfaces [6] may enable a more efficient design, but that is out of scope of this work.

5.2 How Many States for an Element?

We chose elements that can be in only one of two states. But we could have chosen a design that offers greater control. Ideally we would be able to control the exact phase and amplitude with which each element reflects its energy. In terms of our model in equation 2, we would have been able to set any $\mathbf{b}^{(i)} \in \mathbb{C}, |\mathbf{b}^{(i)}| \leq 1$, instead of being restricted to $\mathbf{b}^{(i)} \in \{0,1\}$. Denote the amount of energy that can be directed by the surface in the two cases as h_{IDEAL} and h_{REAL}. In the ideal system, we would be able to align the phases of all $\mathbf{h}^{(i)}$ to get $|h_{IDEAL}| = \sum_i |\mathbf{h}^{(i)}|$. We show here that $|h_{REAL}| \geq |h_{IDEAL}|/\pi$.

Here $|h_{REAL}|$ and $|h_{IDEAL}|$ include only the signal due to the surface ($|\mathbf{h}\cdot\mathbf{b}|$) and not the rest of the environment (h_Z). To maximize signal strength, we would need to align the phases with h_Z too. If we assume that phases are random (assumption 1), we can also prove the result when $|h_{REAL}|$ and $|h_{IDEAL}|$ include h_Z. We discuss this as a corollary of lemma 3 in Appendix A.

Theorem 2. *Under assumptions 2 and 3,* $|h_{REAL}| \geq \frac{|h_{IDEAL}|}{\pi}$ *as* $N \to \infty$.

Proof. Define $A = \int_{-\pi}^{\pi} \sum_{i=1}^{N} |\Re(\mathbf{h}^{(i)} \cdot e^{-j\theta})| d\theta$. $\Re(\cdot)$ denotes the real part of a complex number. The variable A expresses the sum of components of $\mathbf{h}^{(i)}$ along angle θ and integrates over all θ. Each θ corresponds to a perpendicular to a half-plane, as discussed before in Lemma 1. At least one of these, θ_0, corresponds to the optimal half-plane, wherein the optimal solution contains all the $\mathbf{h}^{(i)}$ such that $\Re(\mathbf{h}^{(i)} \cdot e^{-j\theta_0}) > 0$. These contribute $\mathbf{h}^{(i)}$ toward h_{REAL}. Thus,

$$|h_{REAL}| \geq \sum_{i=1}^{N} \max\left\{0, \Re\left(\mathbf{h}^{(i)} \cdot e^{-j\theta_0}\right)\right\} \geq \frac{1}{2}\sum_{i=1}^{N}\left|\Re\left(\mathbf{h}^{(i)} \cdot e^{-j\theta_0}\right)\right| \tag{5}$$

The first inequality holds because the absolute value (LHS) is greater than the real component (RHS). The second inequality holds because otherwise we could have chosen $\pi + \theta_0$ and obtained a better $|h_{REAL}|$. Hence $|h_{REAL}| \geq \frac{1}{2}\max_{\theta\in[\pi,\pi)} \sum_{i=1}^{N} |\Re(\mathbf{h}^{(i)} \cdot e^{-j\theta})| \geq \frac{1}{2}\frac{1}{2\pi}\int_{-\pi}^{\pi}\sum_{i=1}^{N} |\Re(\mathbf{h}^{(i)} \cdot e^{-j\theta})|\, d\theta = \frac{A}{4\pi}$, since the maximum is greater than the average.

Separately, we can rearrange the sum as $A = \sum_{i=1}^{N} \int_{-\pi}^{\pi} |\Re(\mathbf{h}^{(i)} \cdot e^{-j\theta})|\, d\theta = \sum_{i=1}^{N} |\mathbf{h}^{(i)}| \left(\int_{-\pi}^{\pi} |\cos\theta|\, d\theta\right)$. The second step is possible, because $\cos\theta$ expresses the dot product of $\mathbf{h}^{(i)}$ over a unit complex number with phase θ. Since we are integrating over all angles, it doesn't matter which angle we start from. Now we can evaluate the integral to get $A = 4\sum_{i=1}^{N} |\mathbf{h}^{(i)}| = 4|h_{IDEAL}|$. Since $|h_{REAL}| \geq \frac{A}{4\pi}$, $|h_{REAL}| \geq \frac{|h_{IDEAL}|}{\pi}$. □

Alternate designs. We could design an alternate system where $\mathbf{b}^{(i)} \in \{-1, 1\}$ instead of $\mathbf{b}^{(i)} \in \{0, 1\}$. That is, the elements either reflect signal directly or with a 180^o phase difference. In this case, we'd get a $2/\pi$-approximation, which is better than the $1/\pi$ factor we get now. This is because we would have had $\max\{\Re(\mathbf{h}^{(i)} \cdot e^{j\theta_0}), \Re(\mathbf{h}^{(i)} \cdot e^{-j\theta_0})\}$, which removes the $1/2$ factor in equation 5. On the other hand, while our setup can function as both a mirror and a lens (i.e. it can be selectively transparent), this alternate design can only operate as one of the two. A four-state system could potentially offer the benefits of both, a good topic for future work.

5.3 Our Design

Our surface design consists of a panel of metal rectangles of size $\lambda/4 \times \lambda/10$ as shown in Figure 6, where λ denotes

[3]These are called "electrically small antennas".

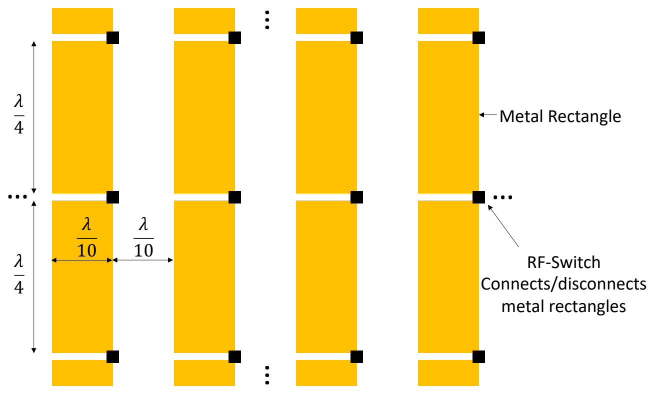


Figure 6: Schematic of the design of our antenna array. This array of rectangles continues in both directions.

the wavelength. They are connected by RF switches, which determine whether or not the rectangles are connected (the switches are placed off-center for practical PCB-design reasons). This design works only for vertically polarized radiation. It can be generalized to all polarizations by having an identical copy perpendicular to this one.

There are two principles of operation. First, the rectangles by themselves are small and hence interact weakly with radiation. If the switches are all off, the surface would be semi-transparent. However, when an RF switch is turned on, it joins two rectangles to form a $\lambda/2 \times \lambda/10$ rectangle. This forms a half-dipole antenna and interacts strongly with incident radiation. We made the strips wide to support a wider bandwidth of operation, since the width allows more modes of oscillation.

Second, if a plate of metal has small holes in it, then radio behaves as if the holes weren't there. This effect is used in Faraday cages, such as those in microwave ovens. A common rule-of-thumb says that the holes need to be smaller than $\lambda/10$, the size of the gaps between rectangles. When switches in adjacent columns are turned on, their rectangles will behave as a continuous sheet of metal, rather than individual columns, almost completely reflecting incident waves. Because neighbors act in a simple way, we expect that the neighbors' state wouldn't change the phase of the currents induced in the rectangle, only the magnitude. Hence the linear model in Equation 2 is approximately correct.

The above reasoning is merely the conjecture that motivated our design. We conduct two experiments to partially validate it. §6.2.1 demonstrates that the linear model is approximately correct, and §6.2.2 shows that the surface can significantly change its opacity. Validating this design in an anechoic chamber would offer more insights. We leave that for future work.

6 Evaluation

6.1 Experiment Setup

Our antennas are fabricated on custom printed circuit boards, with 40 antennas per board. We mount 80 of these boards on two frames and place it next to a wall in our lab. Since metal would interfere with the antennas, the frames are partially built from acrylic. The boards are connected in series with a single SPI serial-to-parallel bus composed of shift registers, allowing our Raspberry Pi controller to serially set the state of each individual element.

To set element state, the controller pushes values through the shift registers at 1 Mbit/s. Higher speeds are impeded by distortion due to the long wires in our setup. Hence pushing a new random state to the surface is the most time-consuming part of measuring it. To alleviate this problem, instead of pushing 3200 bits for each state, we push just 8 bits at a time, which shifts all the bits in the surface by 8 positions. Though the new state isn't completely fresh, each element still gets a new random value that is independent of the previous one. If we were to solve for $\mathbf{h}^{(i)}$ from the system of linear equations, the measurement matrix is still full rank. Hence we expect Algorithm 1 to work. This modification reduces the mean time to measure a state from 7 ms to 0.22 ms. Better engineering will improve performance further. To measure the channel, we use a USRP programmable radio operating in the 2.4 GHz ISM band.

Measurement method. The results in the main evaluation (Figures 10 and 11) are reported as the ratio of the signal strength when RFocus is in its optimized state to the strength when the surface is not present (i.e., *it is physically removed*, not just set to off). We measure this ratio as follows. Let R_I be the improved signal strength, R_0 be the signal strength when the surface is in the all-zeros state (i.e. it is semi-transparent), and R_B be the baseline signal strength when RFocus isn't present. We separately measure R_I/R_0 and R_0/R_B and multiply them to obtain R_I/R_B as desired.

After optimizing the state, we measure R_I/R_0 100 times by repeatedly flipping between the two states every 5 ms and calculating the mean improvement. We find that 1σ error is $< 2\%$ in all cases. To measure R_0/R_B, we first measure R_0. Then we remove the surface and measure R_B. We do this over ≈ 10 transmitter positions in a neighborhood around the original point, and use the median R_0/R_B to correct R_I/R_0. The median helps account for the *systematic* effects of the RFocus surface. It ignores random fading effects that may occur since the RFocus surface, people, or objects will have moved in the meanwhile. The mean absolute correction we had to introduce was 0.6 dB, while the maximum was 1.5 dB.

Total prediction error	Error due to noise
5.4%	2.0%

Figure 7: A linear model predicts the channel due to a state with 5.4% accuracy. If the surface were perfectly linear, the error would have been 2.0% due to noise. Hence the RFocus is approximately, but not fully, linear.

6.2 Microbenchmarks

6.2.1 Linearity

Our analysis of the optimization algorithm (see §4.3) assumes that the elements are not coupled with each other. Hence we used a linear model formalized in Equation (2), where $\mathbf{h}^{(i)}$ are independent of the states of the other elements. We wouldn't expect elements that are far away from each other (say, more than a wavelength) to couple strongly, especially since the elements are flat and radiate perpendicularly away from the surface. However, we packed the elements tightly to more densely cover the surface and beamform more precisely. We hypothesized that the non-linearity due to any resulting coupling will be small (§5.3). We now test this hypothesis.

We prepare several random "test-triples" of states of the form $(\mathbf{b}_A, \mathbf{b}_B, \mathbf{b}_{AB})$. Let (h_A, h_B, h_{AB}) be the corresponding channels. The states are drawn uniformly from the triples where $\mathbf{b}_A \& \mathbf{b}_B = 0$ and $\mathbf{b}_{AB} = \mathbf{b}_A | \mathbf{b}_B$; here states are represented as bit-strings and & and | are bitwise operators. That is, no element is on in both $\mathbf{b}_A$ and $\mathbf{b}_B$, and any element that is on in either $\mathbf{b}_A$ or $\mathbf{b}_B$ is also on in $\mathbf{b}_{AB}$.

If the linear model in Equation (2) is correct, then $h_A/h_Z + h_B/h_Z - 1 = h_{AB}/h_Z$. If an element's effect on the channel ($\mathbf{h}^{(i)}$) depends on whether its neighbors are on, this relation will not hold, since many more pairs of neighbors are on in $\mathbf{b}_{AB}$ than in $\mathbf{b}_A$ or $\mathbf{b}_B$. Since the neighborhoods are substantially different, $\mathbf{h}^{(i)}$ will be different for different states if our assumption is wrong.

Hence, to test linearity, we measure $(|h_A/h_Z|, |h_B/h_Z|, |h_{AB}/h_Z|)$ for several random states and test our ability to predict $|h_{AB}/h_Z|$ given $|h_A/h_Z|$ and $|h_B/h_Z|$ using the above relation. Using $|\cdot|$ doesn't invalidate the relation since for any random state X, $h_X/h_Z = |1 + \Delta h_X/h_Z| \approx 1 + \Re(h_X/h_Z)$ since $\Delta h_X \ll h_Z$. We measure each state 100 times. If Equation (2) were correct, the error in prediction due to measurement noise would have been 2.0%. Our error is 5.4% as shown in Figure 7. We conclude that although RFocus is not perfectly linear, the nonlinearities are small. They *are*, however, large enough that linear regression on $|h_X/h_Z|$ doesn't produce a good predictor.

6.2.2 Controllability and Bandwidth

We expect that the RFocus surface can control its opacity to radio. To test this, we kept the surface in between two wide-bandwidth directional (Vivaldi) antennas pointed at each other.

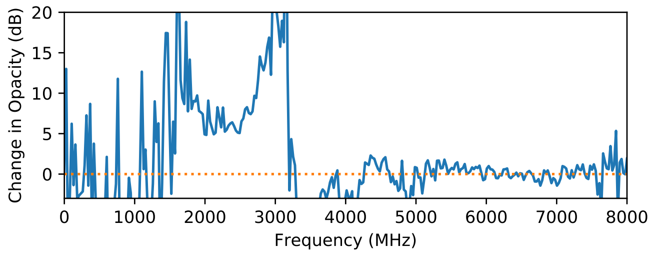


Figure 8: The ability to control the surface's opacity to radio as a function of frequency. This also indicates RFocus's bandwidth of operation.

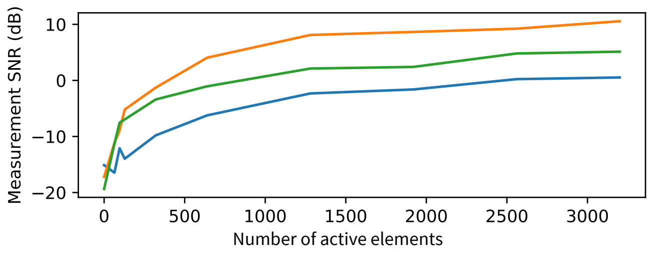


Figure 9: Measurability of the effect random configurations of the RFocus surface on the channel, as a random subset of them are deactivated, for three representative transmitter locations. It demonstrates why boosting (§4.1) is important for RFocus to function.

Using a Vector Network Analyzer (VNA), we compare the signal strength between the antennas when all the elements are turned on to when they are all turned off. We expect that, when the elements are all on, the surface will be much more opaque to radiation, reflecting a large fraction of it. The ratio of signal strength in these configurations is shown in Figure 8: it is consistently greater than 6 dB between 1600 and 3100 MHz. Hence, RFocus can change its opacity by over 75% over a large bandwidth. The peak is closer to 3000 MHz, where the change is well over 10 dB (90% control). But all of our other results are in the 2450 MHz ISM band, in order to conform to FCC rules. Frequency of operation can be tuned by scaling the sizes of the components. (Antenna design is not the focus of this paper.)

The y-axis is cropped at -3 dB for clarity in showing our frequency range of interest. The change falls after 3000 MHz because our RF switch is only rated up to that frequency. At < 1500 MHz, the rectangles, even after joining, are too small to interact with radiation.

6.2.3 Measurability

To find a good configuration, the controller needs to measure $\mathbf{h}$. However, the effect of each individual element, $\mathbf{h}^{(i)}$, is tiny and hard to measure. To aid measurement, we vary all elements randomly and at once (§4.1). This gives us an $O(\sqrt{N})$ boost in

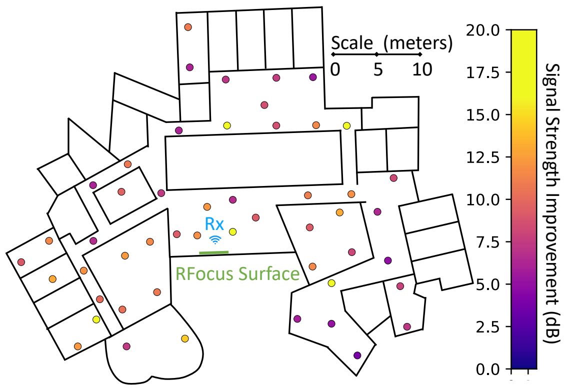


Figure 10: Ratio (in dB) of signal strength without and without the surface. The colored circles represent the improvement when the transmitter was placed at those locations, with the receiver and RFocus placed as shown. The CDF of improvements are shown in Figure 11.

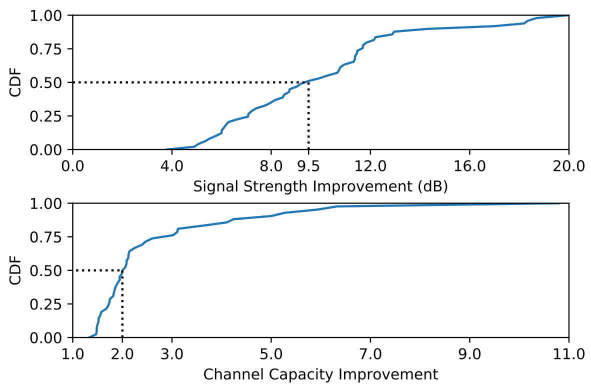


Figure 11: CDF of Improvement in the signal strength and channel capacity

the change in the channel amplitude ($O(N)$ in signal strength), which is easier to measure. To experimentally study the effect of this boost, we compute the Signal to Noise Ratio (SNR) of the *change in the channel due to a random state* as a function of the number of elements in the array. We artificially reduce the size of our array, by deactivating a random subset of it.

We choose 165 different random configurations, and measure the RSSI-ratio for each configuration ≈ 1000 times. The definition of "measurability SNR" is as follows. The "signal" in SNR is the variance in the average RSSI-ratio across all configurations, and the "noise" is the average variance in the RSSI-ratio measurements within each individual configuration. With the receiver and RFocus surface in the locations shown in Figure 10, we place the transmitter at three representative locations covering the full range of signal strengths.

Figure 9 shows the measurability SNR as a function of the number of active elements for these points. We can see that SNR is increases when more elements are varied. Note that when only a few elements are varied, the SNR is very low. This is why boosting the signal by varying all elements at once is critical, especially when the transmitter is far from the receiver.

The impact of random configurations on the channel is still small, because elements can interfere destructively. An optimized state eliminates destructive interference and produces an $O(N)$ effect on the channel ($O(N^2)$ in signal strength). Hence it produces significant gains in signal strength, even though a random state has little impact.

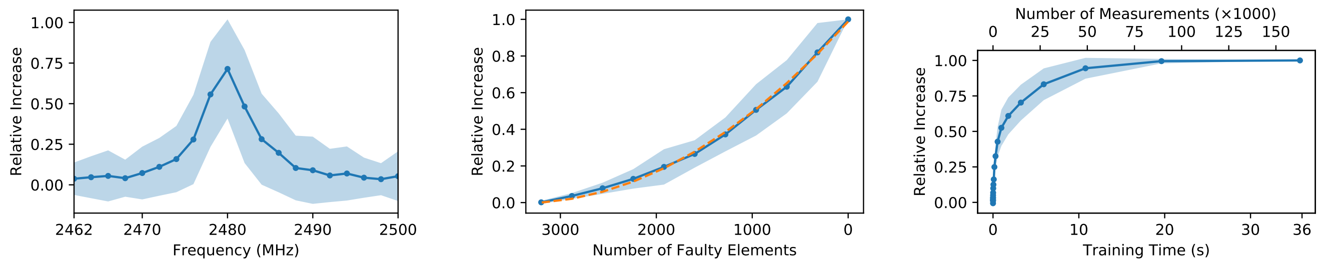
6.3 Signal Strength Optimization

We placed the receiver and the RFocus surface at a constant location as marked in Figure 10. Then we placed the transmitter at various positions in an indoor environment (our lab), and ran the optimization algorithm to maximize signal strength at the receiver. We measure the ratio of the improved signal strength to the signal strength when the surface is physically removed. We plot these on a map in Figure 10. The corresponding CDFs are shown in Figure 11. RFocus provides benefits throughout an entire floor of our building. The minimum, median and maximum improvements across all locations are 3.8 dB, 9.5 dB, and 20.0 dB, respectively.

The five points with the highest improvement had among the smallest unimproved RSSI. Further, they appear in apparently random locations, which makes us postulate that their unimproved RSSI is low due to fading, where destructive interference drastically decreases signal strength. Due to its size, RFocus is unlikely to suffer from fading at all its antennas. Hence, RFocus provides significant benefits for weak channels in such situations.

RFocus's large area allows it to focus energy from the transmitter to the receiver. This is particularly helpful when the transmitters are power constrained, since even a "whisper" will be "heard" clearly at the receiver. Yet, interference does not increase because the transmit power isn't increased. This could enable a new regime of low-power, high throughput IoT sensor devices. Whenever a sensor wants to transmit, it can ask the controller to tailor the surface for that particular endpoint (using a brief high-power transmission). Then it can make its high-bandwidth transmission at low power. Since sensors tend to be stationary, the same trained configuration can be used for extended periods of time. We evaluate this scenario in 6.4.1. Once trained, setting a state takes ≈ 3 ms.

Alternately, radios can use the additional signal strength to increase their channel capacity to obtain higher bitrates, as shown in Figure 11. Channel capacity increases logarithmically with signal strength, and hence the improvement decreases with increasing transmitting power. We set our transmitter to the highest power level (≈ 20 dBm) and adjust the receiver's gain accordingly to avoid saturation.



(a) Improvement for different frequencies, when RFocus was optimized for 2480 MHz. (b) Improvement when some antennas are faulty and flip randomly. Dotted line shows the best quadratic fit. (c) Improvement as a function of training time (and number of measurements)

Figure 12: To understand the trends as we vary different parameters, we plot the mean ± standard deviation of the improvement across all locations pictured in Figure 10. To normalize for differences in absolute improvement for different locations, we plot $\frac{I-1}{Max(I)-1}$. I is the ratio of the improved signal strength to the baseline strength. We normalize relative to the maximum improvement (across all values of the parameter) for that location. Note: $I = 1$ means no improvement. Hence we subtract 1 in our metric, so that positive values denote an improved channel.

6.4 Understanding the Improvement

Neighboring frequencies. Our optimization algorithm only seeks to improve the signal strength at a single frequency (we leave generalizing to multiple frequencies as future work). Nevertheless, we find that it also provides improvement in a 10 MHz neighborhood as shown in Figure 12(a). Multi-path rich environments often have very different signal strength for different frequencies, due to fading [30]. To maximize gain, RFocus could preferentially improve frequencies with the lowest signal strength.

In some cases the frequency that improved the most is different from the target frequency. This could be because that frequency had a much lower baseline signal strength in the un-improved channel due to fading. Hence, the center frequency doesn't have a value of 1. Far away from the target frequency, sometimes the signal strength decreases slightly due to the surface.

Fault tolerance. RFocus is intended to be a large array of inexpensive elements. Hence individual elements are expected to fail, and the system must be robust to failure. To test this, we artificially make a random subset of elements flip randomly, and not as per the controller's instructions, during both training and testing. Figure 12(b) shows the improvement against the number of 'faulty' elements.

Our model of the system suggests that the signal strength increases quadratically with the number of elements (§3.2). This is because each element contributes linearly to the channel amplitude, and the signal strength is the square of the amplitude. Under this model, we expect the normalized mean plotted to have the form $\alpha n^2 + \beta n$, where n is the number of non-faulty elements, and (α, β) are constants. This is the best-fit line pictured.

Note that, this does not imply that gain increases quadratically with the area of the surface, since a larger surface will have more elements that are far away from the source, and hence a smaller $\mathbf{h}^{(i)}$. In the above experiment, we don't change the distribution of $\mathbf{h}^{(i)}$ since we disable a random subset of elements. Hence, the quadratic model works.

Optimization speed. To understand the rate at which the optimization progresses, we plot the signal strength improvement as a function of the training time/number of measurements. As shown in Figure 12(c), 50% of the improvement occurs within 1 sec, with 4500 measurements and 90% improvement occurs within 10 seconds. Note, RFocus has 3200 elements, and we'd expect to need at-least 3200 measurements before the problem is well determined, even ignoring noise. RFocus exploits additional measurements to contend with the fact that measurements are noisy (§6.2.3). RFocus provides some (smaller) benefit with fewer than 3200 measurements, since, at any point in time, RFocus has a hypothesis state that it believes is optimal.

6.4.1 Stability Across Time

Once optimized, how long does an optimized state work? How often do we need to re-optimize our state? To test this, we keep our transmitter in our office on a typical workday, where around 10 people work. We optimized the state at 10AM and measured the improvement due to *that* state till 4PM that day. The top graph in figure 13 shows this. The initial improvement, shown as a dotted line, was 12.6 dB for about 3 minutes before it increased (without re-optimizing the state) for the next ≈ 20 minutes, and then decreased again. We also do a 100 second experiment (in a different location), where two people walk between the endpoints and the surface, intentionally trying to block the direct paths. The bottom graph in figure 13 shows this. Note, the people move only in the middle portion. At other times, the improvement reverts to the baseline.

RFocus's optimized state is robust to motion in the

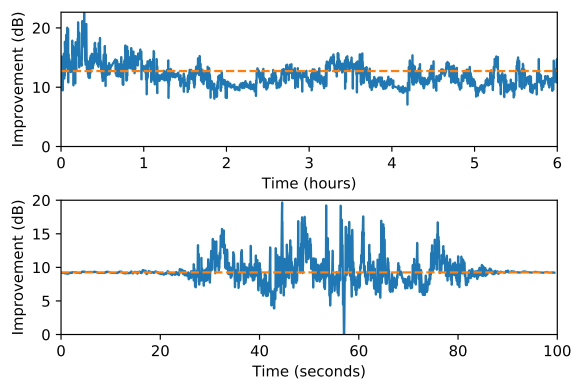


Figure 13: Improvement across time due to a state optimized in the beginning of the time window. This shows that the same optimized state can be used for a long time. The dotted line shows the improvement immediately after the state was optimized. Note the unit change in the x-axis (hours/seconds).

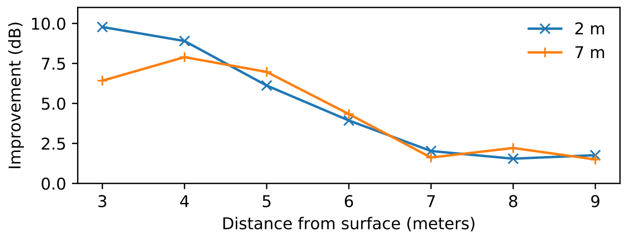


Figure 14: Signal strength improvement when *both* transmitter and receiver are at increasing distances from the surface. In other experiments, the receiver stays close to the surface. The radios endpoints are 2 m and 7m away from each other, in front of the surface, on the same side.

environment. Since optimization takes only 1 to 10 seconds, endpoints can benefit from RFocus for a large fraction of time. RFocus does not support continuous motion and needs to re-optimize the state if the endpoints move.

6.4.2 Moving both endpoints away from the surface

RFocus works best when either (or both) the transmitter and receiver is close to the surface. Performance degrades when both are moved away. Figure 14 shows that RFocus provides benefits up to a few meters from either transmitter or receiver. If ubiquitously deployed, such as on a wall, floor, or ceiling in every room, RFocus allows small devices to realize the benefits of a large antenna array. Alternately, surfaces could be placed close to APs or embedded in building fixtures.

7 Discussion

Efficiency In theory, an N-antenna phased array can achieve an $N\times$ gain in signal strength. RFocus achieves a median $10\times$ gain with 3200 antennas. RFocus suffers from two sources of loss in efficiency. First, only a fraction of the transmitted energy reaches the surface[4]. Second, RFocus can only direct a fraction of the energy that reaches it. This can be improved with better antenna design (§5.2) and metasurfaces [6].

RFocus's gain is qualitatively different from that of a 10 antenna phased array. RFocus precisely focuses a fraction of the transmitted energy, whereas the phased array coarsely directs all of the transmitted energy. Hence, near the receiver, RFocus's beam is more spatially concentrated while the phased array's coarse beam will interfere with nearby endpoints. Nevertheless, we believe these methods are complementary. Phased arrays can direct signals toward RFocus, which can then focus it precisely onto the receiver.

Wireless Controller In our prototype, elements are powered and controlled with wires. This isn't necessary. We envision a design where each element is powered and controlled wirelessly, like a passive RFID tag. The controller acts like a RFID controller that sets the state of each element. RFocus's fault tolerance allows deployment with failure-prone RFID tags (§6.4).

In such a setup, buildings could prefabricate their walls with RFocus elements. Carpets and wallpapers could be sold with RFocus elements already embedded in them. Users can separately buy a controller to control and obtain the benefits of the elements already present in the environment.

8 Conclusion

This paper presented RFocus, a system that moves beamforming functions from the radio transmitter to the environment. RFocus includes a two-dimensional surface with a rectangular array of simple elements, each of which contains an RF switch. Each element either lets the signal through or reflects it. The state of the elements is set by a software controller to maximize the signal strength at a receiver, using a majority-voting-based optimization algorithm. The RFocus surface can be manufactured as an inexpensive thin wallpaper, requiring no wiring. Our prototype implementation improves the median signal strength by $9.5\times$, and the median channel capacity by $2.0\times$.

Human Safety. Because RFocus doesn't emit any energy of its own, it does not increase the total radiation. It cannot focus energy to an area smaller than the size of a wavelength, since any device's ability to focus energy to an area smaller than a wavelength drops exponentially with distance from it. Hence RFocus is no riskier than being near the transmitter.

Ethics Statement. This work raises no ethical concerns.

[4]In most of our experiments, RFocus subtends $\approx 0.4\pi$ steradian on the receiver (equivalently, the transmitter). Hence it controls $\approx 10\%$ of the energy.

9 Acknowledgements

We thank Dinesh Bharadia, Peter Iannucci, Zach Kabelac, Dina Katabi, Colin Marcus, Vikram Nathan, Hariharan Rahul, Deepak Vasisht and Guo Zhang for useful discussion. We would also like to thank members of CSAIL's NetMIT group for letting us borrow their radio equipment.

References

[1] G. B. Airy. On the diffraction of an object-glass with circular aperture. *Trans. of the Cambridge Philosophical Society*, 5:283, 1835.

[2] I. Al-Naib and W. Withayachumnankul. Recent progress in terahertz metasurfaces. *Journal of Infrared, Millimeter, and Terahertz Waves*, 38(9):1067–1084, 2017.

[3] V. Arun and H. Balakrishnan. Rfocus: Practical beamforming for small devices. *arXiv preprint arXiv:1905.05130*, 2019.

[4] C. A. Balanis. Arrays: Linear, planar and circular. In *Antenna theory: analysis and design*, chapter 6. John wiley & sons, 2016.

[5] J. Chan, C. Zheng, and X. Zhou. WiPrint: 3D Printing Your Wireless Coverage. In *HotWireless*, 2015.

[6] H.-T. Chen, A. J. Taylor, and N. Yu. A review of metasurfaces: physics and applications. *Reports on progress in physics*, 79(7):076401, 2016.

[7] H.-T. Chen, A. J. Taylor, and N. Yu. A review of metasurfaces: physics and applications. *Reports on progress in physics*, 79(7):076401, 2016.

[8] M. Chen, M. Kim, A. M. Wong, and G. V. Eleftheriades. Huygens' metasurfaces from microwaves to optics: a review. *Nanophotonics*, 7(6):1207–1231, 2018.

[9] L. J. Chu. Physical limitations of omni-directional antennas. *Journal of applied physics*, 19(12):1163–1175, 1948.

[10] J. Costantine, Y. Tawk, S. E. Barbin, and C. G. Christodoulou. Reconfigurable antennas: Design and applications. *Proceedings of the IEEE*, 103(3):424–437, 2015.

[11] P. del Hougne, M. Fink, and G. Lerosey. Optimally diverse communication channels in disordered environments with tuned randomness. *Nature Electronics*, 2(1):36–41, 2019.

[12] M. Di Renzo and J. Song. Reflection probability in wireless networks with metasurface-coated environmental objects: An approach based on random spatial processes. *arXiv preprint arXiv:1901.01046*, 2019.

[13] P. Genevet and F. Capasso. Holographic optical metasurfaces: a review of current progress. *Reports on Progress in Physics*, 78(2):024401, 2015.

[14] J. Huang and J. A. Encinar. *Reflectarray antennas*, volume 30. John Wiley & Sons, 2007.

[15] N. Kaina, M. Dupré, G. Lerosey, and M. Fink. Shaping complex microwave fields in reverberating media with binary tunable metasurfaces. *Scientific reports*, 4(1):1–8, 2014.

[16] Z. Li, Y. Xie, L. Shangguan, R. I. Zelaya, J. Gummeson, W. Hu, and K. Jamieson. Towards programming the radio environment with large arrays of inexpensive antennas. In *NSDI*, 2019.

[17] C. Liaskos, S. Nie, A. Tsioliaridou, A. Pitsillides, S. Ioannidis, and I. Akyildiz. A new wireless communication paradigm through software-controlled metasurfaces. *IEEE Communications Magazine*, 56(9):162–169, 2018.

[18] C. Liaskos, S. Nie, A. Tsioliaridou, A. Pitsillides, S. Ioannidis, and I. Akyildiz. A novel communication paradigm for high capacity and security via programmable indoor wireless environments in next generation wireless systems. *Ad Hoc Networks*, 87:1–16, 2019.

[19] C. Liaskos, A. Tsioliaridou, S. Nie, A. Pitsillides, S. Ioannidis, and I. Akyildiz. Modeling, simulating and configuring programmable wireless environments for multi-user multi-objective networking. *arXiv preprint arXiv:1812.11429*, 2018.

[20] L. Lipson and H. Lipson. Optical physics. page 340, 1998.

[21] J. S. McLean. A re-examination of the fundamental limits on the radiation q of electrically small antennas. *IEEE Transactions on antennas and propagation*, 44(5):672, 1996.

[22] P. F. McManamon, P. J. Bos, M. J. Escuti, J. Heikenfeld, S. Serati, H. Xie, and E. A. Watson. A review of phased array steering for narrow-band electrooptical systems. *Proceedings of the IEEE*, 97(6):1078–1096, 2009.

[23] P. F. McManamon, P. J. Bos, M. J. Escuti, J. Heikenfeld, S. Serati, H. Xie, and E. A. Watson. A review of phased array steering for narrow-band electrooptical systems. *Proceedings of the IEEE*, 97(6):1078–1096, 2009.

[24] L. Novotny. *Lecture Notes on Electromagnetic Fields And Waves*. 2013.

[25] D. Parker and D. C. Zimmermann. Phased arrays–Part I: Theory and Architectures. *IEEE Trans. on Microwave Theory and Techniques*, 50(3):678–687, 2002.

[26] D. Parker and D. C. Zimmermann. Phased Arrays–Part II: Implementations, Applications, and Future Trends. *IEEE Trans. on Microwave Theory and Techniques*, 50(3):688–698, 2002.

[27] Y.-X. Ren, R.-D. Lu, and L. Gong. Tailoring light with a digital micromirror device. *Annalen der physik*, 527(7-8):447–470, 2015.

[28] L. Subrt and P. Pechac. Intelligent walls as autonomous parts of smart indoor environments. *IET Communications*, 6(8):1004–1010, 2012.

[29] X. Tan, Z. Sun, J. M. Jornet, and D. Pados. Increasing indoor spectrum sharing capacity using smart reflect-array. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2016.

[30] D. Tse and P. Viswanath. *Fundamentals of Wireless Communication*, chapter 2. Cambridge University Press, 2005.

[31] R. B. Waterhouse, S. Targonski, and D. Kokotoff. Design and performance of small printed antennas. *IEEE Transactions on Antennas and Propagation*, 46(11):1629–1633, 1998.

[32] A. Welkie, L. Shangguan, J. Gummeson, W. Hu, and K. Jamieson. Programmable radio environments for smart spaces. In *HotNets*, 2017.

[33] X. Xiong, J. Chan, E. Yu, N. Kumari, A. Sani, C. Zheng, and X. Zhou. Customizing indoor wireless coverage via 3d-fabricated reflectors. In *4th ACM International Conference on Systems for Energy-Efficient Built Environments*, 2017.

A Proof of Lemma 2

First, we prove another intermediate result.

Lemma 3. *Let $\mathbf{b}_\theta$ be the state such that $\mathbf{b}_\theta^{(i)} = 1$ iff $\Re(\mathbf{h}^{(i)} \cdot e^{-j\theta}) > 0$. If assumption 1 holds, then for any θ, the expected value of $\mathbf{h} \cdot \mathbf{b}_\theta$ is $\frac{e^{j\theta}}{\pi}\sum_i |\mathbf{h}^{(i)}|$.*

Proof.

$$E[\mathbf{h}\cdot\mathbf{b}_\theta] = \sum_i E[\mathbf{h}^{(i)}\mathbf{b}_\theta^{(i)}] = \sum_i E[e^{j\theta}\max^{\Re}\{0, \mathbf{h}^{(i)}\cdot e^{-j\theta}\}]$$
$$= \frac{e^{j\theta}}{2}\sum_i E[\Re(\mathbf{h}^{(i)}\cdot e^{-j\theta})] = \frac{e^{j\theta}}{\pi}\sum_i |\mathbf{h}^{(i)}|$$

Here, $\max^{\Re}$ compares the real component, and chooses $\mathbf{h}^{(i)} \cdot e^{-j\theta}$ if its real component is positive. The second step comes from our definition of $\mathbf{b}_\theta$. The third step notes that half of the values are expected to be non-zero and the sum of the remaining values is expected to have a phase of 0, and hence only the real part remains. For the final step, we compute the expected magnitude of the real part of a complex number with random phase in $(-\pi/2, \pi/2]$, given by the factor $\frac{1}{\pi}\int_{-\pi/2}^{\pi/2}\cos(\theta)d\theta = 2/\pi$. □

A corollary of this is that, if $\phi = \text{Arg}(h_Z)$, $|E[h_Z + \mathbf{h}\cdot\mathbf{b}_\phi]| = |h_Z| + \frac{1}{\pi}\sum_i |\mathbf{h}^{(i)}|$. As discussed in §5.2, in an ideal system where we can choose any $\mathbf{b}^{(i)} \in \mathbb{C}, |\mathbf{b}^{(i)}| \le 1$, the optimal assignment achieves a signal strength of $|h_Z| + \sum_i |\mathbf{h}^{(i)}|$. This gives us the same $1/\pi$ bound on what our system can achieve relative to the ideal when we account for h_Z and are willing to assume that phases are random (assumption 1).

Now, we restate and prove lemma 2 here.

Lemma 2. *Let $\mathbf{b}_{OPT}$ be the optimal assignment that maximizes $|h_Z + \mathbf{h}\cdot\mathbf{b}|$ and $\mathbf{b}_\perp$ be such that the i^{th} component $\mathbf{b}_\perp^{(i)} = 1$ iff $\Re(\mathbf{h}^{(i)} \cdot h_Z^*) > 0$. If assumptions 1 and 3 hold, then as $N \to \infty$, $|H(\mathbf{b}_{OPT})|/|H(\mathbf{b}_\perp)| < 1 + \varepsilon \forall \varepsilon > 0$ with high probability.*

Proof. We will argue that $\mathbf{h}\cdot\mathbf{b}_\theta$ is roughly the same as the expected value (with high probability) for any θ. For any $\varepsilon > 0$ and $\theta, \phi \in (-\pi, \pi]$, let A_θ denote the event that $R_\theta = \frac{|\mathbf{h}\cdot\mathbf{b}_\theta|}{|E[\mathbf{h}\cdot\mathbf{b}_\phi]|} > 1 + \varepsilon$. We now show that for any $\varepsilon > 0$, the probability that $\frac{\text{Max}_{\theta\in(-\pi,\pi]}|\mathbf{h}\cdot\mathbf{b}_\theta|}{|E[\mathbf{h}\cdot\mathbf{b}_\phi]|} > 1 + \varepsilon$ goes to zero as $N \to 0$. Let p denote this probability. Note, there are only N distinct values of $\mathbf{b}_\theta$ (say for $\theta_1, ..., \theta_N$), since it only changes when we include/exclude a new $\mathbf{h}^{(i)}$. Hence we need to take a maximum over only N values. Hence p is the probability that $\exists i$ such that that A_{θ_i} happens. That is, $p = P[\cup_{i=1}^N A_{\theta_i}] \le \sum_{i=1}^N P[A_{\theta_i}] = NP[A_\theta]$, for any θ, since the expression is symmetric in θ (note: we need not assume A_{θ_i} are independent here). Let $\bar{h} = \sum_i |\mathbf{h}^{(i)}|/N$, be the mean of the magnitude individual components . Then $|E[\mathbf{h}\cdot\mathbf{b}_\theta]| = N\bar{h}/\pi$ and $R_\theta = \frac{\pi}{\bar{h}}|\sum_{i=1}^N \mathbf{h}^{(i)} \cdot \mathbf{b}_\theta^{(i)}|/N$. From the central limit theorem and lemma 3, we have

$$R_\theta = \frac{\pi}{\bar{h}}\left|\frac{\bar{h}}{\pi} + \mathcal{N}(0, \sigma^2/N)\right| = \left|1 + \frac{\pi}{\bar{h}}\mathcal{N}(0, \sigma^2/N)\right|$$

where σ is the standard deviation in $\mathbf{b}_\theta^{(i)}\mathbf{h}^{(i)}\cos\theta$ for a random θ and $\mathcal{N}$ is a symmetric complex normal distribution with given mean and variance. Note, σ and $\bar{h}$ are bounded as $N \to \infty$, since assumption 3 tells us8 that $\mathbf{h}^{(i)}$ is bounded. Hence $P[A_\theta] = P\left[\mathcal{N}(0, \sigma^2/N) > \varepsilon\bar{h}/\pi\right]$ goes to zero as $N \to \infty$ for any $\varepsilon > 0$. $p = NP[A_\theta]$ also goes to zero as $N \to \infty$, since $P[A_\theta]$ decreases faster than $1/N$. We can follow a similar argument to show that for any $\varepsilon > 0$, the probability that $\frac{|\text{Min}_{\theta\in(-\pi,\pi]}\mathbf{h}\cdot\mathbf{b}_\theta|}{|E[\mathbf{h}\cdot\mathbf{b}_\phi]|} < 1 - \varepsilon$ goes to zero. And hence for any $\varepsilon' > 0$, the probability that $\frac{|E[\mathbf{h}\cdot\mathbf{b}_\phi]|}{|\text{Min}_{\theta\in(-\pi,\pi]}\mathbf{h}\cdot\mathbf{b}_\theta|} > 1 + \varepsilon'$ goes to zero.

Lemma 1 shows that $\exists\theta$ such that $\mathbf{b}_{OPT} = \mathbf{b}_\theta$. Hence

$$H(\mathbf{b}_{OPT}) = \text{Max}_\theta|h_Z + \mathbf{h}\cdot\mathbf{b}_\theta| \le |h_Z| + |\text{Max}_\theta(\mathbf{h}\cdot\mathbf{b}_\theta)|$$

Also $\mathbf{b}_\perp = \mathbf{b}_\phi$ for $\phi = \text{Arg}(h_Z)$. Hence

$$H(\mathbf{b}_\perp) = |h_Z + \mathbf{h}\cdot\mathbf{b}_\phi| = |h_Z| + |\mathbf{h}\cdot\mathbf{b}_\phi| \ge |h_Z| + \text{Min}_\theta|\mathbf{h}\cdot\mathbf{b}_\theta|$$

The second step is true since the two terms have the same phase as $N \to \infty$. Thus,

$$\frac{H(\mathbf{b}_{OPT})}{H(\mathbf{b}_\perp)} \le \frac{|h_Z| + \text{Max}_\theta|\mathbf{h}\cdot\mathbf{b}_\theta|}{|h_Z| + \text{Min}_\theta|\mathbf{h}\cdot\mathbf{b}_\theta|} \le \frac{\text{Max}_\theta|\mathbf{h}\cdot\mathbf{b}_\theta|}{\text{Min}_\theta|\mathbf{h}\cdot\mathbf{b}_\theta|}$$
$$= \frac{\text{Max}_\theta|\mathbf{h}\cdot\mathbf{b}_\theta|}{E[|\mathbf{h}\cdot\mathbf{b}|]}\frac{E[|\mathbf{h}\cdot\mathbf{b}|]}{\text{Min}_\theta|\mathbf{h}\cdot\mathbf{b}_\theta|} < 1 + \varepsilon$$

with high probability for any $\varepsilon < 0$. This proves the theorem. □

CarMap: Fast 3D Feature Map Updates for Automobiles

Fawad Ahmad
USC

Hang Qiu
USC

Ray Eells
Cal Poly, Pomona

Fan Bai
General Motors R&D

Ramesh Govindan
USC

Abstract

Autonomous vehicles need an accurate, up-to-date, 3D map to localize themselves with respect to their surroundings. Today, map collection runs infrequently and uses a fleet of specialized vehicles. In this paper, we explore a different approach: near-real time *crowd-sourced* 3D map collection from vehicles with advanced sensors (LiDAR, stereo cameras). Our main technical challenge is to find a lean representation of a 3D map such that new map segments, or updates to existing maps, are compact enough to upload in near real-time over a cellular network. To this end, we develop CarMap,[1][2] which finds a parsimonious representation of a feature map, contains novel object filtering and position-based feature matching techniques to improve localization robustness, and incorporates a novel stitching algorithm to combine *map segments* from multiple vehicles for unmapped road segments and an efficient map-update operation for updating existing segments. Evaluations show that CarMap takes less than a second to update a map, reduces map sizes by 75× relative to competing strategies, has higher localization accuracy, and is able to localize in corner cases when other approaches fail.

1 Introduction

Autonomous vehicles use a three-dimensional (3D) map of the environment to position themselves accurately with respect to the environment. A 3D map contains *features* in the environment (§2), and their associated positions. As a vehicle drives, it perceives these features using advanced depth perception sensors (such as LiDAR and stereo cameras), then *matches these to features in the map*, and using the feature positions, triangulates its own position.

Maps need to be updated whenever there are significant changes to the environment. Changes to the environment can impact the set of features visible to a vehicle. For example, road or lane closures due to construction or accidents, parked delivery vans impeding traffic flow, parked vehicles on the road-side, or closures for sporting events can cause the set of features in the map to be different from the set of features visible to the vehicle. This impacts feature matching, and can reduce localization accuracy. Figure 1 quantifies this in a simple scenario. In the image on the left, a street has been closed due to an accident. With an outdated map, a car is unable to position itself; an updated map is necessary for accurate positioning.

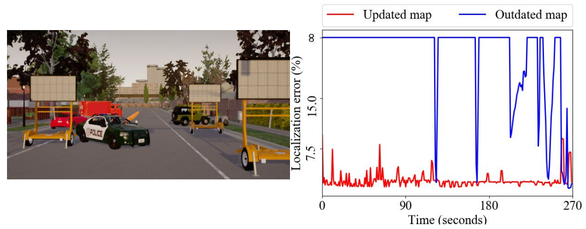


Figure 1: If short timescale events like traffic accidents (left) are not updated in maps, a vehicle cannot localize itself (blue line) because it cannot match the scene with the map. On the other hand, vehicles with updated maps (red line) can localize themselves accurately.

Keeping this map up to date can be tedious. Today, large companies (*e.g.,* Waymo [56], Uber [14], Lyft [12], Here [31], Apple [6], Baidu [7], Kuandeng [11], Mapper [5]) employ fleets of vehicles equipped with expensive sensors (LiDAR, Radar, stereo cameras) and GPS devices. For instance, Apple Map [1] uses vans equipped with a high-precision GPS device, 4 Lidar Arrays, and 8 cameras beside other equipment for capturing mapping data. These vehicles scan neighborhoods periodically with a frequency determined by cost considerations, which could be up to several thousand dollars per kilometer [4]. The scan frequency determines the timescale of environmental changes captured by the map [2]. To capture these changes, vehicle fleets have to continuously traverse the mapped area at very fine timescales [8], which can be prohibitively expensive.

In this paper, we take a first step towards answering the question: *What techniques and methods can ensure near real-time updates to 3D maps?* The most promising architectural approach to this question, which we explore in this paper, is *crowd-sourcing*.[3] In this approach, which leverages the increasing availability of depth perception sensors in vehicles, each vehicle, as it drives through a road segment, uploads *map updates* in near real-time over a cellular network to a cloud service. The cloud service, which acts as a rendezvous point, applies these updates to the map and makes these updates available to other vehicles.

Given today's cellular bandwidths, this architecture is most suitable for a class of 3D maps in which landmarks are *sparse*

[1]https://github.com/USC-NSL/CarMap
[2]Video demo

[3]Incentives for crowd-sourcing are beyond the scope of this paper. Waze has successfully employed crowd-sourcing from vehicles, by providing a navigation service and CarMap can use similar techniques.

features in the environment. Even so, today's feature-based 3D maps of the kind generated by Simultaneous Localization and Mapping (SLAM) algorithms require an order of magnitude higher bandwidth than cellular speeds (§2).

Contributions. Our first contribution (§3) is to identify the most parsimonious representation of feature maps. SLAM feature maps preserve a large number of features, even transient ones, and build indices to enable fast and effective *feature matching*. We show that it is possible to preserve fewer features, and reconstruct the indices, without impacting localization accuracy while reducing map size significantly.

Because our lean map representation throws away information, we have had to re-think feature matching. Our second contribution leverages the observation that, unlike robots, cars have approximate position information (*e.g.,* from GPS). Thus, instead of using statistics of features alone for matching, we also use position information to enable a more robust feature search, leading to improved localization accuracy.

Vehicles will use feature maps over longer time-scales than SLAM maps used by robots,[4] so we must avoid including features (*e.g.,* from parked cars, or pedestrians) that may disappear over those time-scales. We observe that *semantic segmentation* algorithms can identify such features. Our third contribution is a robust resource-aware algorithm that incorporates the semantics of objects in the scene to perform *dynamic object filtering*.

Updates to a map can be of two kinds: *map segments* representing a previously unseen road segment, and *map diffs* representing a transient in a previously-mapped road segment. Our last contribution is a collection of algorithms for map update: a fast and efficient map diff algorithm which generates compact diffs and can integrate these quickly into the map, and a robust map segment *stitching* algorithm that reliably identifies areas of overlap between the map segment and the existing map, and uses features within the overlapped region to transform the segment into the existing map's coordinate frame of reference.

We have embodied these contributions in a system called CarMap. Using experiments on an implementation (§4) of CarMap built upon the top-ranked visual open-source SLAM algorithm [41], and real traces as well as traces from a game-engine simulator [27] we show that (§5): CarMap requires 75× lower bandwidth than competing algorithms; it can generate a map update, disseminate it to a participating vehicle, and integrate the update into the vehicle's map *in less than a second*; its localization accuracy is better than state-of-the-art SLAM algorithms especially when a map is used in dramatically different conditions (*e.g.,* denser traffic) than when it was collected; it can localize a vehicle in some cases when other competitors cannot, such as when a map obtained from one lane is used in another lane in a multi-lane street; its computational overhead is comparable to, and sometimes better than competing strategies; and its feature labeling achieves upwards of 95% accuracy in distinguishing static from non-static objects even when the underlying segmentation algorithms have lower accuracy.

[4]In a robot, SLAM algorithms perform mapping and localization simultaneously. For vehicular use, a SLAM map is collected once, updated intermittently, and used often.

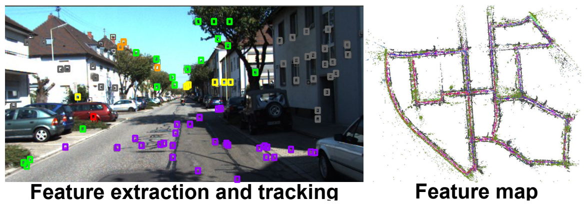


Figure 2: Localization using a feature based map. The picture on the left shows the features in an image, and the picture on the right shows the feature map generated for an area. Features are color-coded by the type of object those features belong to.

2 Background and Motivation

SLAM Principles. SLAM represents a map by a set of *landmarks* and their associated positions [19]. As a vehicle traverses the environment, its sensors (LiDAR, cameras) continuously generate *measurements* of the environment. SLAM continuously outputs (a) detected landmarks, and (b) the current *pose* (position and orientation) of the vehicle. It does this by using maximum *a posteriori* (MAP) estimation [42], finding the landmark position and vehicle pose that best explain the observed measurements.

Feature-based Maps: Terminology. SLAM maps can contain either feature-based landmarks (extracted from cameras [41] or LiDAR [57, 58]) or dense representations such as image frames [28] and occupancy grids [54]. In this paper, we explore crowd-sourcing *feature-based maps* (Figure 2), leaving denser representations for future work[5]. A *feature* is a lower-dimensional representation of some high-dimensional entity in the environment (*e.g.,* a leaf on a tree, or a part of a letter on a roadside sign), and is represented by a feature signature. Features are usually extracted from LiDAR or camera frames. For storage efficiency, SLAM implementations store features from approximately every k frames (so called *keyframes*), for small k. These implementations associate each feature in a keyframe with a relative 3D position with respect to that keyframe. They extract landmarks for the feature-based map from a subset of these features; we call these *map-features*. Maps have a single coordinate frame of reference, and map-features have 3D positions relative to the map's coordinate frame of reference.

SLAM Practice. Practical SLAM implementations are complex (Figure 3) because they have to deal with sensor and estimation errors. We briefly describe SLAM components

[5]Which map technology a vehicle uses is generally proprietary information, but we conjecture, based on anecdotal evidence that lower levels of autonomous driving [43] or vehicles that use stereo cameras will use feature-based maps [9] for cost reasons, while higher-end fully-autonomous vehicles with LiDAR will use denser maps.

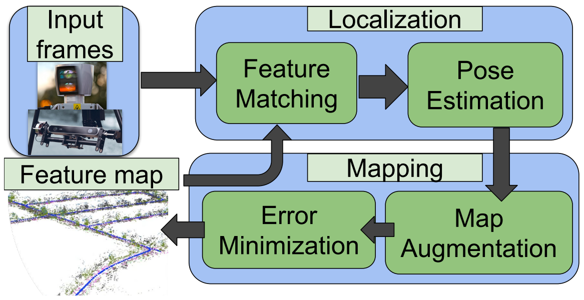


Figure 3: Components of feature-based map generators.

here, and introduce additional background in later sections.

Feature matching. Feature matching (or data association) is the process of matching features in the current frame with features seen in one or more keyframes in the map. SLAM implementations match features in a number of different ways: *e.g.,* image feature matching uses the similarity of image signatures (feature descriptors), and LiDAR 3-D features use feature geometry. Matching is a crucial building block for identifying map-features (as described below). SLAM implementations contain two data structures to speed up feature matching. A *map-feature index* associates map-features to keyframes they occur in. A *feature index* can search for the keyframe whose features most closely match the features in a given frame.

Pose estimation. This component contains algorithms that estimate the pose of the vehicle. As a vehicle traverses an environment, it first extracts features from each frame received from its sensor. Then, the vehicle matches the extracted features with those extracted in the last frame. At this point, the vehicle knows (a) the pose estimate in the previous frame, (b) the positions of the matched features in the previous frame and the current frame. It then uses MAP estimation [42] to estimate the current pose of the vehicle. If the feature matching step does not return enough features to estimate pose accurately, the vehicle uses the feature index to search the entire map for keyframes containing features matching those seen in this frame, a step called *relocalization*.

Map augmentation. Pose estimation can estimate the 3D positions of features in each keyframe. It adds some of these features as map-features, but only after filtering transient features (those that do not occur across multiple frames [41, 58]) or dynamic features (*e.g.,* features that belong to moving vehicles) whose position is not stable across frames.

Error minimization. This component minimizes the error accumulated in the feature map. *Local error minimization* rectifies error accumulation in successive frames using, for example, extended Kalman filters for LiDARs and bundle adjustment [55] for cameras. When vehicles visit a previously-traversed part of the environment, a *loop closure* algorithm finds matches between features in the current frame and features already in the map, then reconciles their position estimates (while also correcting positions of features discovered

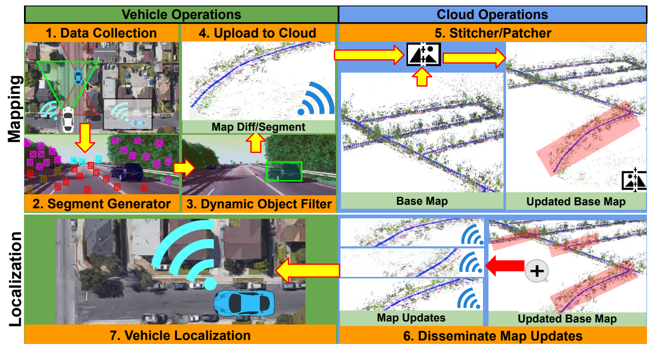


Figure 4: Architecture and workflow of CarMap

within the loop), thereby reducing error.

Challenges. CarMap faces four challenges.

Map size. CarMap could simply upload, over the cellular network, a SLAM map to the cloud, but these maps, which include map-features, keyframes, and the two indices, can be large. A 1 km stretch of our campus generates a 1.5 GB map. A car traveling at 30 kph would require a sustained bandwidth of 100 Mbps, well above achievable LTE speeds [3].[6] Our first challenge is to find a lean map representation that fits within wireless bandwidth constraints.

Environmental dynamics. CarMap maps are meant to be used over a longer timescale than SLAM maps used by robots, so they must be robust to environmental dynamics. For example, if a map includes features from a parked car that has since moved, localization error can increase.

Effective feature matching. As in SLAM, CarMap relies heavily on accurate feature matching for pose estimation, relocalization, and loop closure. However, because CarMap's lean map has less information than SLAM's, its feature matching accuracy can be lower, so CarMap must use a fundamentally different strategy.

Fast map-updates. CarMap must devise fast algorithms to (a) *stitch* additions to the map received from vehicles traversing a previously unmapped road segment (decentralized SLAM algorithms [29] have a similar capability but differ significantly in the details, §6), (b) generate and incorporate changes to the map from temporary obstructions.

3 Design of CarMap

Architecture and Workflow. As vehicles traverse streets (Step 1, Figure 4), they derive lean representations of feature maps using a *map segment generator* that runs on the vehicle (Step 2, §3.1). To this representation, CarMap applies a *dynamic object filter* to improve robustness to environmental dynamics (Step 3, §3.3). CarMap then determines whether this is a new map segment (not available in its own base map). If so, it uploads the entire map segment, else it uploads a

[6]With standard compression techniques (*e.g.,* gzip [26]) the sustained bandwidth is approximately 60 Mbps. Moreover, gzip compression adds latency: it takes approximately 25 seconds to compress a 500MB map collected over 4 minutes.

map diff (Step 4) to a cloud service. The cloud service runs a *stitcher* to add a new segment to the map, or a *patcher* to patch the diff into the existing map (Step 5, §3.4).

A vehicle receives, from the cloud service, segments or diffs contributed by other vehicles (Step 6), *reconstructs* the complete map, and uses it for localizing the vehicle (Step 7, §3.5). Diff generation, stitching, patching, and reconstruction use a *position-based feature index* for feature matching (§3.2), resulting in high feature matching accuracy.

The on-vehicle compute resources needed to run map generation, matching, diff generation, and reconstruction are comparable to those provided by commercial on-vehicle computing platforms like the NVIDIA Drive AGX [13]. CarMap uses (a) cloud storage as rendezvous for map updates from vehicles, and (b) cloud compute to integrate map updates. Extensions to this architecture to use road-side units for storage and processing are left to future work.

3.1 Map Segment Generator

The Problem. As a vehicle traverses a street, it produces map segments. The map segment generator must find the *leanest* representation of the map that respects cellular bandwidth constraints while permitting accurate localization.

As discussed in §2, a complete map contains four distinct components: (A) map-features, (B) features associated with every keyframe, (C) a map-feature index that associates map-features with keyframes used to generate the map-features (recall that a map-feature is one whose position is stable across several keyframes), and (D) a feature index that finds the most similar keyframe to the current frame. Uploading the complete map is well beyond cellular bandwidths (§2).

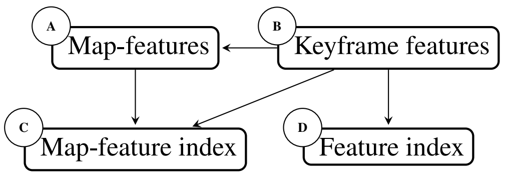


Figure 5: Dependencies between map components

CarMap's Approach. Consider Figure 5 in which an arrow from B to A indicates that B is needed to generate A. Thus, for example, map features are generated from keyframe features. Similarly, to generate the map-feature index, we need both map-features and keyframe features.

From this figure, it is clear that *all other components can be generated from keyframe features*. Thus, in theory, it would suffice for CarMap to upload only the keyframe features, thereby reducing the volume of data to be uploaded. Unfortunately, this does not provide significant bandwidth savings. For a 1 km stretch of a street on our campus (§2), the keyframe features require 400 MB. At 30 kmph, this would require an upload bandwidth of 26.67 Mbps, still above nominal LTE speeds. At higher speeds, CarMap would require proportionally greater bandwidth since the vehicle covers more of the environment (§A.3, Figure 19).

A Lean Map. CarMap uses a slightly non-intuitive choice of map representation: the *map-features* alone. Each map-feature contains the feature signature, the 3D position in the map's frame of reference, and the list of keyframes in which the map-feature appears. In §5, we show that this representation permits real-time map uploads.

Reconstruction. However, to understand why this is a reasonable representation, we describe how one can reconstruct a full SLAM map from these map-features. Map-features have, associated with them, a list of keyframes in which they appear. From these, we can generate keyframe features (a sequence of keyframes and features seen in those keyframes). From these keyframe features, it is possible to generate the feature index and the map-feature index, resulting in the complete SLAM map. §3.5 presents the details.

However, the CarMap map contains *only* map-features whereas a SLAM map contains *all* features seen in every keyframe. These fewer features can potentially impair feature matching accuracy. To address this, CarMap employs a better feature search strategy.

3.2 Robust and Scalable Feature Matching

Background. Feature matching is a crucial component in feature-based localization, and determines both the robustness of feature matching as well as scalability. Feature matching requires two operations: given a frame F, (a) find keyframes with the most similar features, and (b) given a feature f in F and a keyframe K, find those map-features m in K that are most similar to f. The first operation is used in relocalization and loop closure, and the second operation is used for these two tasks as well as fine-grained pose estimation (§2).

Similarity matching. Both of these operations use similarity matching techniques. For example, if a feature is represented by a vector, then, the most similar feature is one closest by Euclidean distance to this feature. Similarly, if a frame F can be represented by a signature in a multi-dimensional space, then the most similar keyframe K is one that is closest by some distance measure.

Scaling similarity matching. To derive scalable feature matching, many SLAM implementations arrange keyframe features in fast data structures. We have used the term feature index in §3.1 to describe these data structures. In practice, implementations construct multiple indices.

To ground the discussion, we take a concrete example from a popular visual SLAM [41] implementation. This implementation discretizes the space of features into hypercubes, and represents each hypercube by a *word*. For example, if a feature f is represented by a vector $< 1, 5 >$, and the hypercube has a side of 10 units, then, f falls into the hypercube defined at the origin. Suppose the hypercube is assigned the word "0". Then, any feature f' that is assigned "0" (*i.e.,* falls into the same hypercube) is close in feature space to f.

Search indices. The two feature matching operations can be implemented in scalable fashion using this word-based discretization. The first operation uses an *inverted index I* that maps each word to all the keyframes that it appears in. To find a keyframe closest to a given frame F, we can use the following algorithm. (i) Map each feature f in F to a word w_f. (ii) For each w_f in F, find all keyframes K associated with w_f in I. (iii) Take the intersection of all keyframes across all words w_f, then find those keyframes whose word histogram is most similar to F. The second operation requires a *word search tree per keyframe K* that maps a given w_f to those features in K that are closest to w_f in feature space.

The Problem. While these data structures work well for indoor robot navigation in relatively static environments, they can fail in more dynamic environments for outdoor vehicles. For example, keyframe word histogram matching can fail when a map's keyframe K was collected from an unobstructed view, while frame F, *taken at the same position*, had a car in front of it which obscured many of the features in K. As another example, consider a map of a 2-lane street where the map was taken from the right lane, but the vehicle using the map is on the left lane; in this case, a feature's signature may change if perceived from a different 3D position and orientation and hence result in a mismatch if the matching is based on feature similarity. In these cases, feature matching can result in *false positives*: a keyframe K far from the vehicle's current position may better match the current frame F than the correct match K' because features at completely different locations in a frame may look visually similar (*e.g.,* features from trees of the same species).

CarMap's Approach. To address these problems, instead of searching all keyframes in the map, CarMap *searches for matches in the vicinity of the vehicle's current position*. CarMap relies on a vehicle's GPS position to scope the search. However, GPS is known to be erroneous, especially in highly obstructed environments [40], so CarMap searches over a large radius around the current GPS position (in our experiments, 50 m, larger than the maximum error reported in [40]).

Keyframe matching. Specifically, in addition to using the inverted index and word histogram similarity to find matching keyframes in the base map, CarMap maintains a global k-d tree [16] of keyframes and uses it to search for all keyframes in the map within a given radius. Then, to localize a vehicle with a frame F in a given map, CarMap uses the GPS coordinates of the vehicle to get all keyframes within a large radius around the GPS position. It then finds the subset of these keyframes that most closely resemble F based on histogram matching. If it cannot find any resembling keyframes, then CarMap uses the keyframes closest to the vehicle's GPS coordinates. For each keyframe K in this subset, CarMap tries to find, for each feature f in F, the closest matching feature in K. To do this, it first performs a coordinate transformation to find the position of f in the map, assuming that F is at K's position, and then performs feature matching.

Feature matching. Based on the position hints of the features, CarMap also maintains another global k-d tree of map features, which partitions 3-D space into different regions to find all features in the map that are closest (by position) to a given feature f. Then, for each feature f in frame F, CarMap finds all map-features that are spatial neighbors, and uses feature similarity to identify the matching features. Using these matching features, it can perform pose estimation. CarMap then attempts to *refine* this pose estimate by searching nearby (in position) map-features for additional feature matches.

3.3 Dynamic Object Filter

Background. As a vehicle traverses an environment, it encounters three types of objects: a) static, b) semi-dynamic, and c) dynamic objects. Static objects are those that are at rest when perceived by the vehicle and are likely to stay in the same position for a long time *e.g.,* roads, buildings, traffic lights, and traffic signs. Dynamic objects are those that are in motion when perceived by the vehicle *e.g.,* moving vehicles and pedestrians. Semi-dynamic objects are those that have the ability to move but might not be in motion when perceived by the vehicle *e.g.,* parked vehicles, construction trucks.

The Problem. SLAM algorithms contain techniques to estimate whether a feature belongs to a dynamic object or not; if it does, that feature is not used in the map (§2). However, for a system designed for vehicles like CarMap, this is insufficient. These techniques work only if the majority of the scene is static and fail in highly dynamic environment (as we show in §5). Similarly, unlike SLAM, CarMap maps are intended to be re-used over longer time scales, during which the environment might change significantly. If a map contains a feature f, say, belonging to a semi-dynamic object such as a parked car which has moved away by the time a vehicle uses that map (before another vehicle has contributed a map diff), keyframe matching and feature matching might fail.

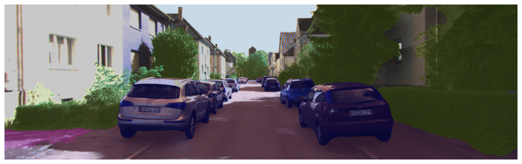


Figure 6: Semantic segmentation of an image while driving.

CarMap's Approach. To counter this, CarMap uses *semantic segmentation* to classify the whole scene into static and (semi-)dynamic objects. Semantic segmentation can be performed on camera data as well as LiDAR data, and refers to the task of assigning every pixel/voxel in a frame a semantic label (Figure 6), such as "car", "building" *etc.* In addition to motion analysis (§2), CarMap leverages these semantic labels to determine whether to add features to the map.

Specifically, CarMap extracts features (Figure 4) and uses semantic segmentation to label each point/pixel in the frame. It then associates each feature with the corresponding semantic label of the particular pixel(s) that the feature covers. As a result, when a feature is generated, besides its feature signature and 3D position, CarMap also appends a semantic label to it. If the semantic label belongs to a dynamic or semi-dynamic[7] object (*e.g.,* car, truck, pedestrian, bike *etc.*), CarMap does not add it to the map.

To detect moving objects we could have used *background subtraction*, but CarMap needs the ability to also detect semi-dynamic objects (*e.g.,* parked cars). Object detectors can generate loose bounding boxes for semi-dynamic objects, which can result in incorrect matches between features and their corresponding objects.

Challenges. Semantic segmentation poses two challenges in practice. First, it is prone to errors, especially at the boundaries of different objects. For example, a state-of-the-art segmentation tool, DeepLabv3+ [20], has an iIoU[8] score of 62.4% on a semantic segmentation benchmark (CityScapes [23]). Second, it uses deep convolutional neural networks that are computationally very expensive (*e.g.,* DeepLabv3+ runs at only 1.1 FPS on a relatively powerful desktop equipped with an NVIDIA GeForce RTX 2080 GPU).

Robust labeling. To tackle the first challenge, CarMap tracks feature labels across multiple frames and uses a majority voting scheme for deriving robust labels. Consider a feature f that is detected and tracked in multiple keyframes (only these features are likely to be added as map-features). In each keyframe, we determine the semantic label associated with the f. Instead of labeling each feature with its semantic label, we perform a coarser classification, determining whether that label belongs to a static (road surface, traffic signals, buildings, and vegetation *etc.*) or a non-static (cars, trucks, pedestrians) *etc.* This coarser classification overcomes boundary errors in segmentation: even if the segmentation algorithm identifies a pixel as belonging to a building when it actually belongs to a tree in front of the building, because both of these are static objects, the pixel would be correctly classified as static. CarMap then does a majority voting across these coarser labels to determine whether f is static or non-static. In §5, we show that this approach results in high classification accuracy.

Resource usage. Semantic segmentation CNNs can run at low frame rates. However, CarMap only needs to determine the label of a feature when creating map-features. These are assessed at keyframes, so, segmentation needs only be applied at keyframes. Depending on the vehicle's speed, SLAM algorithms [41] can generate keyframes at 1-10 frames per second. In §5, we explore a resource/accuracy trade-off: running slightly less accurate, but lower resource intensive CNNs still gives acceptable performance in our setting. When segmentation cannot run on every keyframe, we mark the missed keyframe's features as *unlabeled*.

[7]For brevity, we use the term *dynamic object filter* for this capability, but it can detect semi-dynamic objects as well.

[8]The IoU (intersection over union) metric is biased towards classes covering a large image area. Hence, for autonomous driving, the iIoU metric is preferred which is fairer towards all classes.

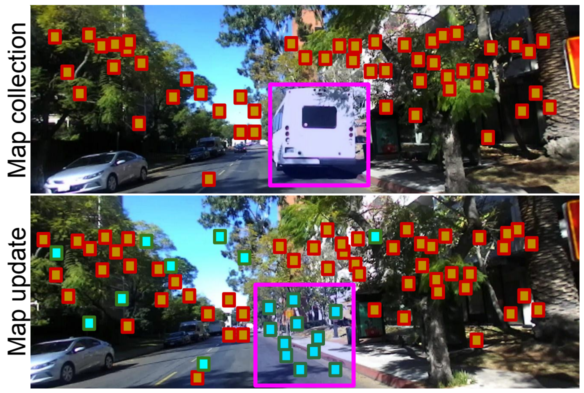


Figure 7: When adding a new region to the base map, the vehicle uploads the whole map segment (above). For updating a existing map segment, CarMap generates a map diff containing new map features (below, new map features marked in blue).

3.4 Map Updater

Map Diffs. When a vehicle traverses a segment that exists in its own map, CarMap generates a compact map diff to report newly discovered map features.

The Problem. CarMap may discover new features for two reasons. In Figure 7, if the feature map were constructed from the top image, a vehicle traveling through the same region at a later time (bottom image) might see new features previously occluded by the bus. Moreover, sparse SLAM algorithms are designed to capture only a small portion of all the features in the environment to ensure real-time operation, so a new traversal may discover additional features (Figure 7).

CarMap's Approach. A *map diff* compactly represents the newly discovered features. To explain how CarMap generates a map diff, consider a vehicle V, traversing a road segment R_A at time t_1, having an on-board map segment M_A of the same area from an earlier time t_0. CarMap loads the on-board map segment M_A into memory and marks all map elements (map-points and keyframes) as pre-loaded elements in the map. As the vehicle V traverses R_A, it localizes itself in the map segment M_A. At the same time, for every feature f_{road} the vehicle perceives, it uses CarMap's robust feature matching (§3.2) to query and match it against features f_A present in the map segment M_A in the same spatial vicinity. If the match is successful, that means the feature is already present in the map. If not, it is a new feature. This yields a set of features f_{diff} and keyframes K_{diff} that have been introduced in the time interval $\delta t = t_1 - t_0$. The vehicle uploads this *diff* map to the cloud service. The cloud service's patcher receives this and patches these map elements (f_{diff} and K_{diff}) into the base map. It also sends out the patch to all vehicles so that they can update their base maps.

Removing features no longer visible is tricky because those

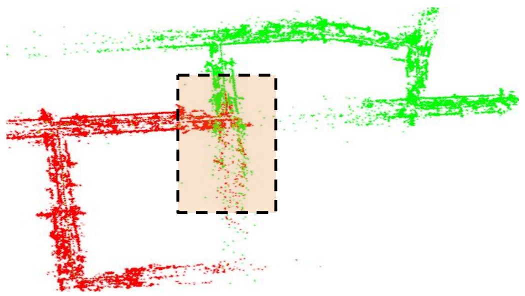


Figure 8: CarMap stitching together two feature maps. The highlighted regions represent overlapped sub-segments.

features could be, for example, occluded by a parked vehicle. It is, however, important to do this in practice (*e.g.,* features from objects present during a transient road closure). We are currently working on a robust algorithm for this.

Map Segment Stitching. When it traverses a previously unseen road segment, CarMap uploads the map segment to a *map stitcher* in the cloud.

The Problem. CarMap's stitcher adds map segments (§3.1) received from vehicles into its *base map* (Figure 8). CarMap must address three challenges while stitching a map segment into a base map. It must efficiently find potential regions of overlap between two map segments. The stitcher only has access to map-features at keyframes whereas SLAM algorithms preserve all features in each keyframe; feature matching can potentially be more difficult in CarMap. To scale well, CarMap must incrementally add new map segments to the base map without recomputing the whole map.

CarMap's Approach. Algorithm A.1 depicts the stitching algorithm. Suppose we have two map segments, the new incoming map segment M_s and the base map M_b. To stitch M_s with the base map M_b, CarMap first reconstructs (lines 4-5) (§3.5) the two map segments (R_b, R_s). Then, it uses (line 6) fast feature search (§3.2) to find the sub-segments (sequences of keyframes) that overlap (O_b, O_s). It then applies (line 7) feature matching between these sub-segments and uses these matches to compute the *coordinate transformation matrix* between M_s and M_b. It uses this matrix to transform M_s into M_b's coordinate frame of reference (lines 8-9). Finally, it removes duplicate features observed in both segments. §A.1 describes some of the details of this algorithm.

3.5 Reconstruction

Map Segment Download. Before a vehicle enters a street, it retrieves a map segment from the cloud service. This segment uses the lean representation described in §3.1.

Reconstruction Details. CarMap places map-features into keyframes, and adds them to the k-d tree structures. It then generates word histograms and per-keyframe word search trees as in SLAM. To do this, it must compute the 2D and 3D positions of each map-feature in the associated keyframe (recall that a map-feature's position in a map segment is with respect to the map's frame of reference). To reconstruct the position of a given map-feature f in a keyframe k, $P_f\left(k\right)$, CarMap uses the global 3D position of the map-feature $P_f\left(O\right)$, the respective keyframe's position $P_K\left(O\right)$ and rotation matrix $R_K\left(O\right)$ to perform an inverse transformation:

$$P_f\left(k\right) = \left[R_K^{-1}\left(O\right) * \left\{P_f\left(0\right) - P_K\left(0\right)\right\}\right] \tag{1}$$

4 Implementation of CarMap

Software we use. We have implemented CarMap by modifying a visual SLAM algorithm, ORB-SLAM2 [41], the top-ranked open-source visual odometry algorithm for mono, stereo, and RGB-D cameras in the KITTI vision-based benchmarks [30] for self-driving cars. At least one other visual SLAM implementation [45] has a very similar implementation structure, so CarMap can be ported to it. It should also be possible to incorporate CarMap ideas into LiDAR SLAM implementations, but we have left this to future work.

For semantic segmentation, we use MobileNetV2 [51], a light-weight version of DeepLabv3+ [20] designed for mobile devices. We use OpenCV [18] for image transformations, the Point Cloud Library (PCL) [50] for point cloud operations, and the C++ Boost library [52] for serializing and transferring the map files over the network.

Our Additions. On top of these, we have added a number of software modules necessary for the six components described in §3. CarMap reuses the feature extraction, index generation, and similarity-based feature matching modules in ORB-SLAM2 (ORB-SLAM2 is 9620 lines of C++ code), but even so, it requires approximately 15,000 additional lines of C++ code. §A.2 discusses these additions in detail.

5 CarMap Evaluation

In this section, we evaluate (a) real-time end-to-end latency of map update using experiments and (b) the localization accuracy of CarMap using trace-driven simulation. We then report on microbenchmarks for its lean map representation, feature map stitching, segmentation, and spatio-temporal robustness in localization, using both synthetic and real-world traces.

5.1 Methodology

Traces. For our end-to-end accuracy evaluations, we use 15 km of stereo camera traces that we curated using CarLA [27], the leading simulation platform for autonomous driving supported both by car manufacturers and major players in the computing industry. CarLA can simulate multiple vehicles driving through realistic environments — the simulator has built-in 3D models of several environments including freeways, suburban areas, and downtown streets. Each vehicle can be equipped with stereo cameras or LiDAR sensors, and the simulator produces a *trace* of the sensor outputs as the cars drive through. When curating our CarLA traces, we model a stereo camera with the same properties (stereo baseline, focal length *etc.*) used in the KITTI dataset. When evaluating

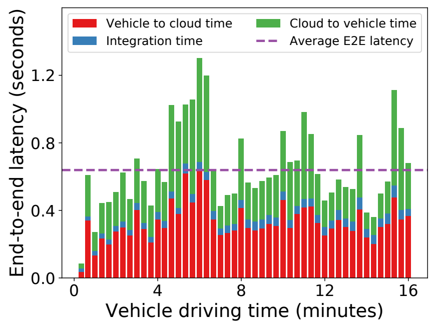


Figure 9: End-to-end latency results for CarMap's map update operation enables real-time map updates (average end-to-end latency is approximately 0.6 seconds.)

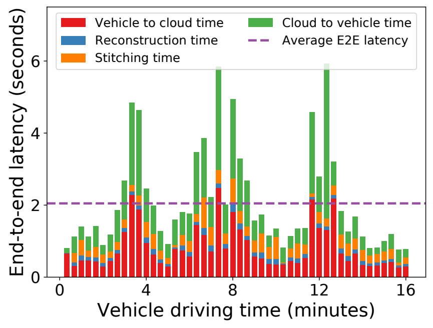


Figure 10: End-to-end latency for CarMap's map stitch operation. The stitch operation, on average, takes approximately 2.0 seconds for unmapped regions.

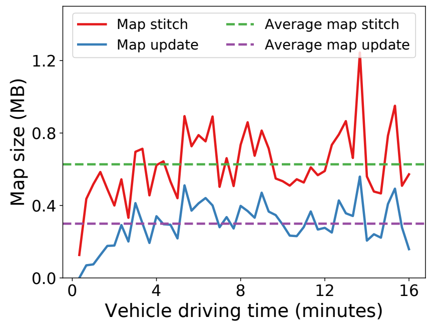


Figure 11: Vehicle map uploads for map stitch and update operations. Map updates reduce required bandwidth by 2x as compared to stitching map segments.

CarMap, we only extract the left and right images from the modeled camera after which we process the frames like we would for a real-world camera. We do not extract depth or segmentation labels from CarLA but instead generate them using ORB-SLAM2's stereo matching and a segmentation CNN respectively.

For some of our microbenchmarks, we also used 22 km of real-world traces from the KITTI odometry benchmark [30]. The KITTI benchmark traces only have a single run for each route, but for our end-to-end evaluations, we need one run to build the map, and another to use the map for localization. This is why we use traces from CarLA.

Finally, to validate real-time map updates (§5.2), we used 8 km of stereo camera data from our campus.

Metrics. For most evaluations, we are interested in end-to-end latency, localization accuracy, and map size. To calculate localization accuracy, we build a map for a region and then localize another vehicle that drives in the same region using that map. In this case, the localization error is the average translational/localization error (used in KITTI odometry benchmark [30]) between the ground truth position of the vehicle and its estimated position, averaged over the whole trace. In some experiments, we also measure compute times for various operations. These measurements were taken on an Alienware laptop equipped with an Intel i7 CPU running at 4.4 GHz with 16 GB DDR4 RAM and an NVIDIA 1080p GPU with 2560 CUDA cores.

Scenarios. For the end-to-end accuracy experiments, we generate CarLA traces to mimic three different kinds of driving conditions: a) suburban streets (light traffic and some parked vehicles), b) freeway roads (dense traffic), and c) downtown roads (dense traffic, with parked vehicles on both sides). For each of these, we generate traces for a static scene (no traffic), and for a dynamic scene (with traffic). This allows us to evaluate maps built for one kind of scene (*e.g.,* static), but used in another (*e.g.,* dynamic).

Comparison. In all these evaluations, we compare the performance of: a) maps generated by ORB-SLAM2, b) a stitched map generated by QuickSketch [15], and c) a stitched CarMap map. QuickSketch is a competing approach to map crowd-sourcing that does not attempt near real-time map updates. In QuickSketch, map segments are raw stereo camera traces, and the stitching algorithm feeds new map elements from the camera trace into an existing base map generated by ORB-SLAM2. QuickSketch uses ORB-SLAM2's relocalization and feature matching components. We repeat each experiment three times and report the average values.

5.2 Near Real-Time Map Updates

Methodology. To measure end-to-end latency of map updates, we drove a vehicle for 16 minutes (8 km) equipped with an Alienware laptop tethered to a phone with an LTE connection. The laptop sends map updates to a remote server which runs CarMap's diff integration and stitching operations, then sends the map updates back to the vehicle. The end-to-end latency includes: update generation and transmission on the sender, update processing on the cloud, and update transmission and integration on the receiver. We conducted two experiments.

Map Diffs. In the first experiment, we measure end-to-end latency when all updates are in the form of map diffs (*i.e.,* the vehicle drives through a previously mapped area). CarMap generates map diffs every 10 s. As Figure 9 shows, the *average end-to-end latency for CarMap's map update operation is 0.6 s*[9]. Update transmission times dominate the cost, since diff integration is fast (§3.4).

Map Segment Stitching. In the second real-time experiment, we measure the end-to-end latency when all updates are in the form of map segments (*i.e.,* the vehicle drives through a previously un-mapped area). As before, CarMap generates map segments every 10 s; in this case, however, the cloud service needs to perform an expensive stitch operation (§3.4).

The overall end-to-end latency for map segment updates in CarMap (Figure 10), although about 3.2× more than map-

[9] As an aside, vehicles rely on these maps only to localize themselves, not for safety-critical operations (for which they use their sensors).

updates, is *still only 2.1 seconds on average*. Two factors contribute to a higher end-to-end latency. First, map segments are about 2-4× larger than map diffs (Figure 11). Second, they require about 10× more computation than map updates. Map update integration is only 50 ms whereas the partial reconstruction (§A.1) and stitching take nearly 500 ms. Even so, transmission and reception times dominate.

In summary, in CarMap, map updates can be made available to other vehicles in under a second. Even in the rare event that a vehicle traverses an un-mapped road segment, map updates can be made available in about 2 s.

5.3 End-to-End Localization Accuracy

We now demonstrate that CarMap has comparable or better localization accuracy than ORB-SLAM2 and QuickSketch [15] for three different scenarios: static scene, dynamic scene, and multi-lane localization.

Static Scene Maps. In this scenario, we build a map from a static scene with no dynamic or semi-dynamic objects (a *static-map*). We then use this map to localize a vehicle that drives in: a) the same static scene (resulting in a *static-trace*), and b) the same scene with parked and moving vehicles (resulting in a *dynamic-trace*). Figure 12 shows the average error and map sizes for each scheme and scenario. (We show the error distributions in Figure A.9 and Figure A.11).

In all three environments (suburbia, downtown, and freeway), the localization error for the static-trace in the static-map shows that CarMap is able to *localize as accurately as ORB-SLAM2* even though the *map sizes are 23-26*× smaller. Similar results hold for CarMap when compared against recent map crowdsourcing work, QuickSketch. This is because CarMap preserves all map-features that contribute most towards accurate localization.

However, for the dynamic-trace on the static-map, CarMap has nearly *28× better localization accuracy* than ORB-SLAM2 and Quicksketch. These differences arise from two features in our scenarios: traffic, and the presence of parked cars, which impact localization accuracy in different ways.

To understand why, consider a dynamic-trace on a suburban street. If the location or number of parked cars in the dynamic-trace are different from those in the static-map, the signature of the observed frame (its word histogram) is different in the trace than in the map. Because ORB-SLAM2 relies on word-histogram matching for re-localization, it fails to find the right keyframe candidates to localize. In contrast, because CarMap filters features belonging to parked cars, the vehicle in the suburban street sees similar features as in the map, and can re-localize more accurately.

Now consider a dynamic-trace on the freeway, in which a vehicle's view can be obscured by other vehicles, so it is unable to observe many of the features in the map. This causes ORB-SLAM2's word histogram matching to fail. CarMap uses all keyframes within a 50 m radius of its current position, so it always has keyframe candidates to search from. Even when histogram matching succeeds, ORB-SLAM2 uses per-keyframe word search trees that can result in false-positive feature matches. CarMap uses feature position based search to avoid this. In this scenario, moreover, ORB-SLAM2 believes features belonging to vehicles moving in the same direction to be stable (since their relative speed is near zero), makes them map-features and uses them to track its own motion. CarMap's dynamic object filter avoids this pitfall.

Dynamic Scene Maps. In this scenario, we build a map from a dynamic scene (a *dynamic-map*) and then use the map to localize in a dynamic- or static-trace. Figure 13 summarizes the results from this experiment (Figure A.10 plots the distribution of mapping errors).

The results for the dynamic-map are more dramatic than those for the static-map. CarMap's map is 15-36× smaller than ORB-SLAM2's or QuickSketch's map. Despite this, these two approaches *fail to localize* (denoted by ∞) on static-traces in downtown and suburban streets. In the static-trace, very few of the perceived features appear in the dynamic-map, and relocalization fails completely. CarMap does well here because it filters out all cars (parked or moving). For the dynamic-trace, its accuracy is nearly 50× better than ORB-SLAM2 and QuickSketch. CarMap's accuracy is lowest for the downtown dynamic-trace (with a 5% translational error) in which parked and moving cars obscure a lot of features in the map, resulting in fewer matches.

Multi-Lane Localization. In this set of experiments, we consider a somewhat more challenging case, for each of our scenarios: building a map by traversing one lane of a multi-lane street (4 freeway lanes, or 2 lanes in the suburban and downtown streets), and then trying to localize the vehicle in each of the remaining lanes. As before, we build both static-maps (Figure 14) and dynamic-maps (Figure 15).

For the freeway static-map, ORB-SLAM2 cannot localize beyond the second lane, while CarMap can localize across all four lanes. For the dynamic-map, a more challenging case, CarMap can localize one lane over, but ORB-SLAM2 and QuickSketch cannot localize at all (denoted by ∞). In all these cases, ORB-SLAM2's search strategy fails because its keyframe search relies on the vehicle's perspective being the same as the map's perspective: in these experiments, that assumption does not hold. CarMap, by contrast, matches features by position not perspective, so is much more robust.

Similar results hold for suburban and downtown streets: ORB-SLAM2 and QuickSketch are unable to localize, but CarMap is able to localize in all cases, with low error.

In §A.4, we show that CarMap's *mapping accuracy*, which measures the inherent error introduced by mapping, is comparable to ORB-SLAM2.

5.4 Other Performance Measures

Map Sizes in Real-World Traces. §5.3 shows that CarMap's maps are lean relative to competing strategies, but these are

Mapping scheme	Freeway			Suburbia			Downtown		
	Static error (%)	*Dynamic error (%)*	*Map size (MB)*	*Static error (%)*	*Dynamic error (%)*	*Map size (MB)*	*Static error (%)*	*Dynamic error (%)*	*Map size (MB)*
ORB-SLAM2	1.06	24.3	157.6	0.60	36.7	101.2	1.06	26.2	320.7
QuickSketch	0.95	24.4	157.6	0.60	37.0	99.1	0.90	22.1	303.0
CarMap	1.09	**2.06**	**5.94**	0.68	**1.01**	**3.94**	1.15	**3.87**	**14.3**

Figure 12: Mapping error and map sizes for a static-map used with static- and dynamic-traces, for each scenario. ∞ indicates that the scheme was not able to localize at all.

Mapping scheme	Freeway			Suburbia			Downtown		
	Static error (%)	*Dynamic error (%)*	*Map size (MB)*	*Static error (%)*	*Dynamic error (%)*	*Map size (MB)*	*Static error (%)*	*Dynamic error (%)*	*Map size (MB)*
ORB-SLAM2	6.33	62.3	110.6	∞	45.4	105.6	∞	25.9	291.1
QuickSketch	19.6	64.7	113.3	∞	44.7	108.4	∞	22.4	267.3
CarMap	**1.39**	**1.5**	**5.25**	**1.22**	**0.86**	**7.2**	**1.39**	**5.05**	**7.91**

Figure 13: Mapping error and map sizes for a dynamic-map used with static- and dynamic-traces, for each scenario. ∞ indicates that the scheme was not able to localize at all.

Mapping scheme	Freeway			Suburbia	Downtown
	2nd Lane	*3rd Lane*	*4th Lane*	*Parallel lane*	*Parallel lane*
ORB-SLAM2	3.79	∞	∞	∞	∞
QuickSketch	4.29	∞	∞	∞	∞
CarMap	**2.26**	**3.52**	**4.85**	**1.96**	**4.03**

Figure 14: Mapping error (%) for multi-lane localization in static environments using maps collected from one lane in other parallel lanes.

Mapping scheme	Freeway	Suburbia	Downtown
	Parallel lane	*Parallel lane*	*Parallel lane*
ORB-SLAM2	∞	∞	∞
QuickSketch	∞	∞	∞
CarMap	**4.80**	**5.24**	**9.81**

Figure 15: Mapping error (%) for multi-lane localization in dynamic environments using a map collected from one lane in other parallel lanes. CarMap is robust to spatio-temporal changes.

Mapping environment	**Mapping scheme**	**Map: dynamic Trace: static**	**Map: static Trace: dynamic**
Suburbia	**QuickSketch**	∞	∞
	CarMap	0.8	1.6
Downtown	**QuickSketch**	∞	∞
	CarMap	0.93	4.91

Figure 16: Mapping errors (m) for stitching map segments from different traffic conditions. CarMap is robust to temporal changes because: a) removes dynamics, and b) robust feature search.

DCNN	**FPS**	**Label segmentation accuracy**	**Object segmentation accuracy**
DeepLabV3+	1.1	76.7	96.6
Tuned DeepLabV3+	2.2	71.6	96.4
MobileNetV2	5	72.8	97.6

Figure 17: Semantic segmentation accuracy for different DCNNs. By classifying labels into static and dynamic objects, the segmentation accuracy for all DCNNs is above 96%.

for synthetically generated traces. Figure 18 shows the map sizes for the 11 real-world KITTI sequences. Across all sequences, CarMap *reduces map size 20×*. About 20% of this savings comes from removing the reconstructible indices (the *No Index* column), and another 60% from removing keyframe-features after generating the indices (the *No Keyframe-features* column).

As a vehicle travels faster, feature maps capture more data from the environment and generate data at a higher rate. We validate this in Figure 19 by calculating the bandwidth requirements of all 11 KITTI traces. Maps generated by ORB-SLAM2 and the No-Index approach are impractical at all speeds for LTE wireless upload and impractical for LTE download at speeds over 40 kph. The No Keyframe-features alternative is impractical for LTE upload at speeds over 60 kph. CarMap requires less than 3 Mbps up to 80 kph (the highest speed in the KITTI traces). Similar results hold for CarLA-generated traces (§A.3).

Other factors determine map size, including visual richness of the environment, lighting, weather *etc.* CarMap's map size should still be an order of magnitude smaller than competing approaches; future work can validate this.

Localization Time. CarMap's accuracy comes at the cost of a slightly higher per-frame localization time. During localization, CarMap's feature search adds overhead. To quantify this, we built a map from a very large trace with 4541 frames and then tried to localize in the same trace. ORB-SLAM2 has a per-frame localization cost of 0.023 s, while CarMap's is only marginally higher (0.033 s).

Map Load Time. When it receives a map segment, CarMap needs to read the segment from disk, reconstruct the keyframe features, and the indices. Figure 20 quantifies the total cost of these operations (called the map *load time*) for each of the 11 KITTI sequences. The load times for other alternatives are normalized by those for CarMap.

Interestingly, except for sequences 00, 01 and 06, load times for CarMap *are less than ORB-SLAM2* (on average, 0.95×). For most sequences, CarMap's load time is lower than ORB-SLAM2 because the latter's map is large enough that the time to load it from disk exceeds CarMap's reconstruction overhead. Other alternatives (*No Index* and *No Keyframe-features*) have large maps and high reconstruction overhead. When CarMap's reconstruction cost is (marginally) higher than ORB-SLAM2, it is because the corresponding scenes have a dense map-feature index, leading to a slightly higher reconstruction cost. (See §A.5 for details). Denser map-feature indices are found in environments with keyframes that have a large number of common map-features (*e.g.,* freeways). We have verified both these observations (equivalent map-load times and slightly higher load times for dense map-feature indices) for CarLA sequences.

Loop Closure. Loop closure is an important component of SLAM systems. For the KITTI dataset, we have verified that, even though its maps contain only map-features, CarMap can perform all loop closures that ORB-SLAM2 can.

5.5 Robustness

Robust Feature Matching. We compare CarMap's feature matching performance to that of ORB-SLAM2's native feature matching approach (we use ORB-SLAM2's default parameters for matching). For this, we build a map segment for a static trace and then use that trace to localize: a) the same static trace, b) a static trace from a parallel lane, c) a dynamic trace from the same lane, and d) a dynamic trace from a parallel lane. We collect the trace using CarLA on a freeway, and use two metrics: a) feature matching ratio (the percentage of map-features matched in the current trace), and b) localization error (m).

Figure 21 shows that for all scenarios, robust feature matching *is able to find more matches and hence results in lower*

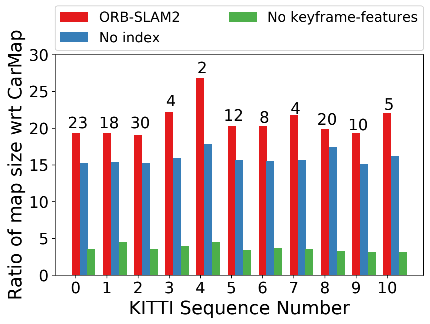


Figure 18: Map sizes on KITTI traces: for each alternative, the map size is normalized by CarMap's map size. The number on top of each group of bars shows the size in MB of CarMap's map for the corresponding KITTI trace. CarMap reduces map size by 20x for unmapped regions.

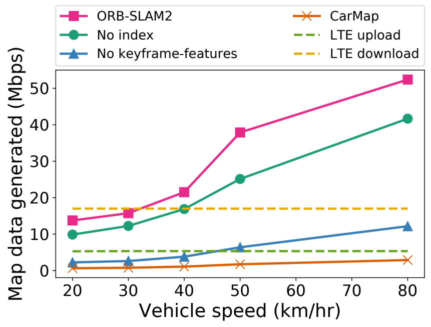


Figure 19: Bandwidth requirements for the four mapping schemes averaged over diverse environments in all 11 KITTI sequences at different speeds. CarMap can support near-real time uploads over LTE at speeds up to 80 kph whereas other schemes fail even at low speeds.

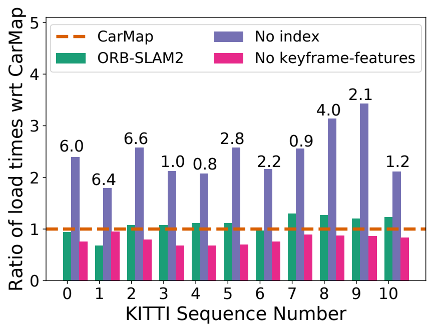


Figure 20: Load times on KITTI traces: for each alternative, the load times are normalized by CarMap's load time (whose absolute value is on top of each group of bars). CarMap loads faster than ORB-SLAM2 (*i.e.,* ORB-SLAM2's load time ratio > 1), except for 3 KITTI sequences.

Scheme	Feature matching (FM) ratio (%)	Localization error (m)
Map: Lane 1 (static) Trace: Lane 1 (static)		
Normal FM	70	0.83
Robust FM	**96**	**0.51**
Map: Lane 1 (static) Trace: Lane 1 (dynamic)		
Normal FM	49	1.4
Robust FM	**66**	**0.57**
Map: Lane 1 (static) Trace: Lane 2 (static)		
Normal FM	52	1.2
Robust FM	**74**	**0.66**
Map: Lane 1 (static) Trace: Lane 2 (dynamic)		
Normal FM	32	1.5
Robust FM	**40**	**0.6**

Figure 21: CarMap's robust feature matching finds more features in different conditions and thus localize better than ORB-SLAM2.

Base map lane	Mapping scheme	Map segment lane			
		1st	2nd	3rd	4th
1st	QuickSketch	0.67	0.90	∞	∞
	CarMap	0.70	**0.88**	∞	∞
2nd	QuickSketch	0.34	0.68	0.90	∞
	CarMap	1.1	**0.62**	0.97	**1.15**
3rd	QuickSketch	∞	5.8	0.68	3.2
	CarMap	**1.13**	**0.96**	**0.63**	**1.19**
4th	QuickSketch	∞	∞	10.83	0.67
	CarMap	∞	**1.0**	**1.0**	**0.66**

Figure 22: Mapping error (m) for multi-lane stitching. CarMap's stitching algorithm uses a more robust feature search based on position hints to stitch map segments two lanes apart where competing strategies fail (∞ shows an unsuccessful stitch operation.)

localization error as compared to ORB-SLAM2's feature matching. The base case (static-map used by a static trace) shows that normal feature matching fails to detect 30% of the features even though the same trace is used for mapping and localization. The introduction of dynamic objects reduces the feature matching ratio because features are occluded by vehicles and hence cannot be detected even with robust matching.

Making Semantic Segmentation Robust. CarMap makes segmentation robust by voting across multiple keyframes, and using a coarser static vs. non-static classification. Figure 17 shows CarMap's overall accuracy, for three different versions of DeepLabv3+. These DNNs are the DeepLabv3+ trained on the CityScape dataset pre-trained, a fined tuned DeepLabv3+ trained on the KITTI dataset and a light-weight version of DeepLabv3+ (MobileNetv2) for mobile devices. The third column shows that CarMap achieves upwards of 96% accuracy if we apply segmentation to every keyframe. Semantic segmentation, by itself, achieves only 70% accuracy in label assignment (second column).

The first column shows the frame rate these DNNs run at. The frame rate needs to be fast enough to process every keyframe, or at worst, every other keyframe (at which segmentation accuracy drops to about 85%, and below which it drops to unacceptable levels, §A.7). In the KITTI dataset, the average across the 11 sequences is 3.17 keyframes per second, well within the rate of the MobileNetv2 version. One of these sequences runs at 10 keyframes per second, so for this sequence MobileNetv2 would process every other keyframe. For more dynamic scenes, it might be necessary to devise faster semantic segmentation techniques, and we expect the vision community will make advances in this direction.

Multi-Lane Stitching. CarMap can stitch map segments collected from different lanes. For this experiment, we collect traces from four parallel lanes on a freeway in CarLA. Using each of these four traces as base maps, we try to stitch map segments from other lanes into it, then evaluate the mapping error for the new maps. Figure 22 shows the absolute mapping errors (in meters) for these stitched map segments. The first column shows the lane used to collect the base map and the last four columns show the absolute mapping error of a stitched map with each of these lanes. The ∞ sign represents a failure to stitch segments from the two lanes.

Although QuickSketch's base map has 20× more features than CarMap and it localizes a stereo camera trace in that base map instead of another map segment (CarMap), it cannot stitch two lanes away. On the other hand, CarMap's stitching algorithm uses robust feature matching (§3.2) and can stitch map segments collected two lanes away (*e.g.,* map segments from lane 1 and lane 3). CarMap's robustness comes purely from using position hints to find the set of key-frames to match, and to find matching map features, while QuickSketch uses ORB-SLAM2's built-in matching methods (in this experiment, we do not compare against ORB-SLAM2 because it does not contain a map stitch operation).

Stitching in Different Traffic Conditions. Besides being robust to spatial changes, crowdsourced map collection and update requires robustness to temporal changes as well (*e.g.,* changes in traffic during different times of day). To evaluate this, we collect stereo camera traces from CarLA in suburban and downtown areas in the same environment during different traffic conditions (no traffic and heavy traffic). Using these traces, we evaluate the ability of the mapping schemes to stitch these map segments by comparing their mapping error.

Figure 16 shows that QuickSketch is unable to stitch because it fails to relocalize a trace in different traffic conditions (§5.3). This, again, is because its stitching is solely based on appearance-based matching whereas CarMap uses position hints as well to make its stitching more robust. By contrast, CarMap is able to stitch map segments collected across different traffic conditions. We evaluate the sensitivity of stitching accuracy to the degree of map segment overlap in §A.6.

6 Related Work

Decentralized SLAM. Decentralized SLAM systems [24] leverage multiple agents to run SLAM in unknown environments. CarMap can be considered an instance of decentralized SLAM [22] with some differences. In decentralized SLAM, the agents (robots) have limited compute-power and only run visual odometry [29]. This leads to inaccurate localization whereas vehicles in CarMap localize more accurately because they run both mapping and localization. Decentralized SLAM sends all keyframe features to a central collector which performs all mapping operations [53] whereas CarMap only sends map-features to a cloud service to ensure real-time map exchanges. Similarly, in decentralized SLAM, the collector finds overlap between maps of different agents using the histogram word approach, does not remove environmental dynamics and hence is not robust like CarMap. Decentralized SLAM [47] uses features from a single keyframe overlap to compute the transformation matrix whereas CarMap is more robust and uses features from multiple keyframes.

Visual SLAM. Although we have implemented CarMap on top of ORB-SLAM2 [41], our study of other SLAM systems shows that it can be easily ported to other keyframe-based visual SLAM algorithms like S-PTAM [45]. In future work, we can extend CarMap to group features into higher-dimensional planes [32] to further improve localization accuracy. As wireless speeds increase, it might be possible to design over-the-air map updates for dense mapping systems like [38] using techniques similar to ours. We have left this to future work.

Long Term Mapping. Our implementation uses traditional computer vision-based features (ORB [49]) to build the map, but these can be replaced with better, more stable CNN-based features [25]. After running a feature extractor, CarMap uses motion tracking and semantic segmentation to select stable features to build the map. Mask-SLAM [33] proposes a similar dynamic object filter to CarMap but CarMap uses majority voting and robust labeling to account for limited on-board computational resources and boundary segmentation errors. Other approaches [17, 34] remove dynamic features from multiple maps collected along the same trace using background subtraction. Even the most static features are not persistent for larger timescales. Future work for longer timescale mapping can integrate CarMap with a persistence filter presented in [48] that estimates the life period of a feature based on an environmental evolution model. CarMap benefits from map-element culling techniques [35] that scale maps sizes by the scale of the environment rather than the number of miles driven. Mobileye [10] crowdsources collecting 3D maps for vehicles using monocular cameras whereas CarMap is designed for 3D sensors like LiDARs, and stereo cameras.

Vehicle Sensing and Communication. LiveMap [21] uses GPS and monocular cameras to automate road abnormality detection (*e.g.,* pothole detection). With its depth perception capabilities, CarMap can more accurately position roadside hazards. AVR [46] extends vehicular vision using feature maps and would benefit from CarMap. Although the bandwidth requirements for CarMap are within the LTE speeds today, it can benefit from systems [36] that schedule redundant transmissions over multiple networks. Recent work in object detection on mobile devices [39] introduces a fast object tracking method that can be used in CarMap to enable faster segmentation. For stitching map segments from rural, unmapped regions, CarMap can benefit from [44] which enables autonomous navigation in such areas.

7 Conclusion

CarMap enables near real-time crowd-sourced updates, over cellular networks, of feature-based 3D maps of the environment. It finds a lean representation of a feature map that fits within wireless capacity constraints, incorporates robust position-based feature search, removes dynamic and semi-dynamic features to enable better localization, and contains novel map update algorithms. CarMap has better localization accuracy than competing approaches, and can localize even when other approaches fail completely. Future work can explore LiDAR sensors, mapping over timescales in which even relatively static features can disappear, dense map representations, infrastructure-based sensing for map updates in low vehicle density areas, and automated update of semantic map overlays (accidents, available parking spots).

Acknowledgements. Our shepherd Kyle Jamieson and the anonymous reviewers provided valuable feedback. The work was supported by grants from the US National Science Foundation (Grant No. CNS-1330118) and General Motors.

References

[1] Apple Is Rebuilding Maps From the Ground Up. https://techcrunch.com/2018/06/29/apple-is-rebuilding-maps-from-the-ground-up/, 2018.

[2] Here Self-Healing Maps. https://go.engage.here.com/self-healing.html, 2018.

[3] State of Mobile Networks: USA - OpenSignal. https://opensignal.com/reports/2018/07/usa/state-of-the-mobile-network, 2018.

[4] The Golden Age of HD Mapping for Autonomous Driving. https://medium.com/syncedreview/the-golden-age-of-hd-mapping-for-autonomous-driving-b2a2ec4c11d, 2018.

[5] There's No Google Maps for Self-Driving So This Startup Is Building It. https://www.technologyreview.com/s/612202/theres-no-google-maps-for-self-driving-cars-so-this-startup-is-building-it/, 2018.

[6] Apple Maps Image Collection. https://maps.apple.com/imagecollection/, 2019.

[7] Baidu. https://www.baidu.com/, 2019.

[8] Carmera. https://www.carmera.com/fleets/, 2019.

[9] GM's Hands-free Driving Feature to Work on 70,000 Additional Miles of Highways This Year. https://www.theverge.com/2019/6/5/18653628/gms-super-cruise-hands-free-driving-feature-highway-milage, 2019.

[10] HERE and Mobileye: Crowdsourced HD Mapping for Autonomous Cars. https://360.here.com/2016/12/30/here-and-mobileye-crowd-sourced-hd-mapping-for-autonomous-cars/, 2019.

[11] Kuandeng. http://www.kuandeng.com/html/1/index.html, 2019.

[12] Lyft Level 5. https://level5.lyft.com/, 2019.

[13] NVIDIA Drive AGX. https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/, 2019.

[14] Upgrading Uber's 3D Fleet. https://medium.com/uber-design/upgrading-ubers-3d-fleet-4662c3e1081, 2019.

[15] Fawad Ahmad, Hang Qiu, Xiaochen Liu, Fan Bai, and Ramesh Govindan. QuickSketch: Building 3D Representations in Unknown Environments using Crowdsourcing. In *2018 21st International Conference on Information Fusion (Fusion)*, pages 2314–2321. IEEE, 2018.

[16] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, September 1975.

[17] Julie Stephany Berrio, James Ward, Stewart Worrall, and Eduardo Nebot. Identifying Robust Landmarks in Feature-based Maps. *arXiv preprint arXiv:1809.09774*, 2018.

[18] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[19] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, Jose Neira, Ian Reid, and John J. Leonard. Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-perception Age. *Trans. Rob.*, 32(6):1309–1332, December 2016.

[20] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with Atrous Separable Convolution for Semantic Image Segmentation. In *ECCV*, 2018.

[21] Kevin Christensen, Christoph Mertz, Padmanabhan Pillai, Martial Hebert, and Mahadev Satyanarayanan. Towards a Distraction-free Waze. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 15–20. ACM, 2019.

[22] Titus Cieslewski, Siddharth Choudhary, and Davide Scaramuzza. Data-efficient Decentralized Visual Slam. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2466–2473. IEEE, 2018.

[23] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.

[24] Alexander Cunningham, Manohar Paluri, and Frank Dellaert. Ddf-sam: Fully Distributed Slam using Constrained Factor Graphs. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3025–3030. IEEE, 2010.

[25] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superpoint: Self-supervised Interest Point Detection and Description. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 224–236, 2018.

[26] P. Deutsch. RFC1952: GZIP File Format Specification Version 4.3, 1996.

[27] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An Open Urban Driving Simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[28] Jakob Engel, Thomas Schöps, and Daniel Cremers. LSD-SLAM: Large-scale Direct Monocular Slam. In *European conference on computer vision*, pages 834–849. Springer, 2014.

[29] Christian Forster, Simon Lynen, Laurent Kneip, and Davide Scaramuzza. Collaborative Monocular Slam with Multiple Micro Aerial Vehicles. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3962–3970. IEEE, 2013.

[30] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are We Ready for Autonomous Driving? the Kitti Vision Benchmark Suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361. IEEE, 2012.

[31] Here. The Self-healing Map From Here. https://go.engage.here.com/self-healing.html, 2019.

[32] Mehdi Hosseinzadeh, Yasir Latif, and Ian Reid. Sparse Point-plane Slam. In *Australasian Conference on Robotics and Automation 2017 (ACRA 2017)*.

[33] Masaya Kaneko, Kazuya Iwami, Toru Ogawa, Toshihiko Yamasaki, and Kiyoharu Aizawa. Mask-SLAM: Robust Feature-based Monocular SLAM by Masking using Semantic Segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 258–266, 2018.

[34] B Ravi Kiran, Luis Roldao, Beñat Irastorza, Renzo Verastegui, Sebastian Süss, Senthil Yogamani, Victor Talpaert, Alexandre Lepoutre, and Guillaume Trehard. Real-time Dynamic Object Detection for Autonomous Driving using Prior 3d-maps. In *European Conference on Computer Vision*, pages 567–582. Springer, 2018.

[35] Henrik Kretzschmar, Giorgio Grisetti, and Cyrill Stachniss. Lifelong Map Learning for Graph-based SLAM in Static Environments. *KI*, 24:199–206, 09 2010.

[36] HyunJong Lee, Jason Flinn, and Basavaraj Tonshal. Raven: Improving Interactive Latency for the Connected Car. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 557–572. ACM, 2018.

[37] Shiqi Li, Chi Xu, and Ming Xie. A Robust O (n) Solution to the Perspective-n-point Problem. *IEEE transactions on pattern analysis and machine intelligence*, 34(7):1444–1450, 2012.

[38] Yonggen Ling and Shaojie Shen. Building Maps for Autonomous Navigation using Sparse Visual Slam Features. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 1374–1381. IEEE, 2017.

[39] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge Assisted Real-time Object Detection for Mobile Augmented Reality. In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*. ACM, 2019.

[40] Xiaochen Liu, Suman Nath, and Ramesh Govindan. Gnome: A Practical Approach to NLOS Mitigation for GPS Positioning in Smartphones. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, page 163–177, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Raul Mur-Artal and Juan D Tardós. ORB-SLAM2: An Open-source Slam System for Monocular, Stereo, and Rgb-d Cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[42] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[43] Society of Automotive Engineers International. Automated Driving Levels of Driving Automation Are Defined in New SAE International Standard J3016. (2014), 2014.

[44] Teddy Ort, Liam Paull, and Daniela Rus. Autonomous Vehicle Navigation in Rural Environments Without Detailed Prior Maps. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2040–2047. IEEE, 2018.

[45] Taihú Pire, Thomas Fischer, Gastón Castro, Pablo De Cristóforis, Javier Civera, and Julio Jacobo Berlles. S-ptam: Stereo Parallel Tracking and Mapping. *Robotics and Autonomous Systems*, 93:27–42, 2017.

[46] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. AVR: Augmented Vehicular Reality. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (Mobisys)*, MobiSys '18, pages 81–95, Munich, Germany, 2018. ACM.

[47] Luis Riazuelo, Javier Civera, and JM Martínez Montiel. C2tam: A Cloud Framework for Cooperative Tracking and Mapping. *Robotics and Autonomous Systems*, 62(4):401–413, 2014.

[48] David M Rosen, Julian Mason, and John J Leonard. Towards Lifelong Feature-based Mapping in Semi-static Environments. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1063–1070. IEEE, 2016.

[49] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An Efficient Alternative to Sift or Surf. 2011.

[50] Radu B Rusu and S Cousins. Point Cloud Library (pcl). In *2011 IEEE International Conference on Robotics and Automation*, pages 1–4, 2011.

[51] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted Residuals and Linear Bottlenecks. In *CVPR*, 2018.

[52] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[53] Patrik Schmuck and Margarita Chli. Multi-uav Collaborative Monocular Slam. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3863–3870. IEEE, 2017.

[54] Lu Sun, Junqiao Zhao, Xudong He, and Chen Ye. DLO: Direct Lidar Odometry for 2.5d Outdoor Environment. *2018 IEEE Intelligent Vehicles Symposium (IV)*, Jun 2018.

[55] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle Adjustment—a Modern Synthesis. In *International workshop on vision algorithms*, pages 298–372. Springer, 1999.

[56] Waymo. Building Maps for a Self-driving Car. https://medium.com/waymo/building-maps-for-a-self-driving-car-723b4d9cd3f4, 2016.

[57] Ji Zhang and Sanjiv Singh. LOAM: Lidar Odometry and Mapping in Real-time. In *Robotics: Science and Systems*, volume 2, page 9, 2014.

[58] Ji Zhang and Sanjiv Singh. Visual-lidar Odometry and Mapping: Low-drift, Robust, and Fast. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2174–2181. IEEE, 2015.

A Appendix

A.1 Map Stitching Details

Algorithm A.1 describes the details of the stitching algorithm. The following two paragraphs discuss two key aspects of stitching.

Finding Overlap. To find potential regions of overlap, CarMap uses two strategies. When the cloud service receives the new map segment M_s, it uses the GPS positions and word-histograms associated with M_s to coarsely find potentially overlapping keyframes in the base map M_b. For this, CarMap reconstructs all the data structures in M_b and only word-histograms and keyframe-features of M_s using the methods described in §3.5.

Then, CarMap finds a finer-grained overlap O_b and O_s (granularity level of map-points) between M_s and M_b. For this, CarMap uses the reconstructed keyframe features of M_s. For each keyframe k_s in O_s, it uses the k-D tree to find *all* features (§3.2) in O_b that match features in k_s, instead of only matching features belonging to the two overlapping keyframes k_s and k_b. At the end of this process, there is a pairwise matching of features between O_b and O_s.

```
Input  : Base map $M_b$ and new map segment $M_s$
Output: Stitched base map $M'_b$
1  if $M_b$ is empty then
2      $M'_b \leftarrow M_s$;
3  else
4      $R_b \leftarrow$ Reconstruct($M_b$);
5      $R_s \leftarrow$ PartialReconstruct($M_s$);
6      $O_b, O_s \leftarrow$ FindOverlap($R_b$,$R_s$);
7      $T_{bs} \leftarrow$ FindTransform($O_b$, $O_s$);
8      $M^\star_s \leftarrow T_{bs} * M_s$;
9      $M'_b \leftarrow$ Merge($M_b$, $M^\star_s$);
10 end
```

Algorithm A.1: Stitching Algorithm

Computing the transformation matrix. In the next step, CarMap computes the transform (translation and rotation) to re-orient and position M_s in M_b. To do this, it finds the keyframe k_s from the new map segment with the maximum number of matched features from the previous step. Then it uses a perspective n-point (PnP [37]) solver to derive the coordinate transformation matrix, then transforms each map feature in M_s to M_b's frame of reference. After the transformation, CarMap removes all the duplicate map-features in the overlapping region O_b of the resulting base map M'_b that originated as a result of the transformation.

A.2 Implementation Details

The following paragraphs describe how we have implemented CarMap components on top of ORB-SLAM2.

Map segment generator. This component takes the output of ORB-SLAM2 (which includes map-features, keyframe features, and the two indices), and simply strips all other components other than the map-features. We have also added the ability to periodically transmit complete *map segments*. On the receiver, we added a module to reconstruct (§3.5) the keyframe features from the received lean map, and re-generates the indices.

Fast feature search. For this, we added the k-D tree data structure, and associated code for manipulating the tree and searching in the tree, and re-used ORB-SLAM2 code for re-positioning a feature in a keyframe.

Stitcher. Stitching functionality does not exist in ORB-SLAM2. For stitching maps, we wrote our own modules for ORB-SLAM2. We also added support for finding overlapped keyframes and computing the transformation matrix.

Map updater. We wrote our own module for map updates. At the vehicle, our map update module uses a fast feature search for finding differences in the two feature sets (environment and base map). At the cloud, the module integrates these differences into the base map.

Dynamic object filter. We added a dynamic object filter to the mapping component of ORB-SLAM2 which invokes semantic segmentation and applies majority voting to decide the label associated with each map feature.

Map exchange. We added another module to allow the exchange of *map segments*, map updates, and the base map between the vehicles and the cloud service.

A.3 Bandwidth Requirements

Map Size with Change in Speed. As a vehicle's speed increases, it sees more features and hence generates larger maps. As such, we generated CarLA traces in which we increased the speed of the vehicle while keeping time constant. The goal of this experiment is to see if CarMap's maps can stay within the wireless bandwidth limits at different speeds. Figure A.1 shows that CarMap's maps are well below the wireless bandwidth limits today by a large margin and this is not true for competing strategies. ORB-SLAM2 and the No index approach's maps cannot be uploaded over current wireless networks at all speeds and cannot be downloaded for speeds greater than 10 kmph. The No keyframe-features approach is also infeasible for LTE upload for speeds over 15 kmph. We also validated this in Figure 19 for real-world traces from the KITTI dataset.

Bandwidth Savings with Map Updates. In this section, we evaluate the ability of CarMap's update operation to reduce the amount of bandwidth required to update the base map. For these experiments, we collected traces from the same area in CarLA in three different traffic conditions *i.e.,* static with no parked vehicles, semi-dynamic with only parked vehicles and dynamic with both parked and moving vehicles. We build a map for each traffic condition and then measure the amount of bandwidth required to update the existing map with features

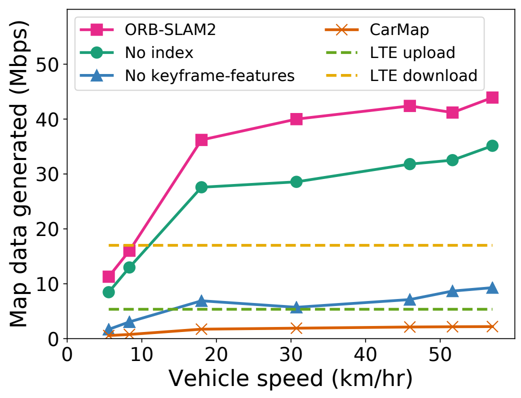


Figure A.1: Bandwidth requirements for mapping schemes at different speeds in CarLA. The bandwidth required to upload CarMap maps are well below the LTE upload limits.

Pre-loaded map	Trace	Map stitch size (MB)	Map update size (MB)
Static	Static	0.81	0.21
Static	Semi-dynamic	0.80	0.35
Static	Dynamic	0.82	0.29
Semi-dynamic	Static	0.81	0.27
Semi-dynamic	Semi-dynamic	0.80	0.08
Semi-dynamic	Dynamic	0.82	0.27
Dynamic	Static	0.81	0.24
Dynamic	Semi-dynamic	0.80	0.25
Dynamic	Dynamic	0.82	0.12

Figure A.2: Bandwidth requirements for map updates in CarMap under different traffic conditions

from a different set of conditions. The baseline we compare is with the map stitch case in which we would upload the whole map segment to the cloud service and the cloud service would only add the new map elements to the map.

The results from the experiment (Figure A.2) show that, given a base map of the area, map updates can reduce the amount of bandwidth required to integrate new features in the base map by 4-10$\times$ compared to sending the whole map segment (75$\times$ savings as compared to QuickSketch and ORB-SLAM2). This happens because the map update only sends new features whereas the map stitch sends the whole perceived map segment.

A.4 Mapping Accuracy

In this section, we evaluate how CarMap's reduced map sizes affect localization accuracy. For this experiment, we use all 11 real-world traces from the KITTI dataset. We generate maps for each of these traces, use them as base maps and localize the same trace in these maps. We compare the generated trajectory with the ground truth positions. Figure A.3 shows the average localization error divided by the length of the whole sequence for all the KITTI sequences. Even though CarMap reduces map sizes by a factor of 20, it is able to localize as accurately as ORB-SLAM2 in almost all KITTI sequences because: a) it preserves the most important map elements (map-features), and b) robust feature matching.

KITTI sequence number	Average translational error over whole trace (%)			
	ORB-SLAM2	No index	No keyframe-features	CarMap
00	0.16	0.17	0.17	0.16
01	0.69	0.69	0.68	0.74
02	0.18	0.17	0.17	0.18
03	0.31	0.32	0.32	0.36
04	0.33	0.31	0.39	0.40
05	0.07	0.07	0.07	0.08
06	0.12	0.21	0.20	0.20
07	0.14	0.15	0.14	0.15
08	0.33	0.30	0.26	0.30
09	0.32	0.32	0.30	0.29
10	0.46	0.50	0.50	0.48

Figure A.3: Localization error for CarMap over all KITTI sequences. Even though CarMap uses 20X fewer features in its map, its localization error is almost the same as ORB-SLAM2.

Total distance covered (m)	Average translational error (%)			
	ORB-SLAM2	No index	No keyframe-features	CarMap
50	3.62	3.61	3.75	2.70
100	2.25	2.15	2.22	1.85
150	1.55	1.54	1.60	1.50
200	0.88	1.22	1.27	1.30
300	0.73	0.91	0.94	0.99
400	0.66	0.75	0.78	0.80
600	0.56	0.60	0.62	0.58
800	0.53	0.55	0.53	0.49
1000	0.30	0.30	0.31	0.31
2000	0.32	0.30	0.30	0.32
3000	0.24	0.23	0.22	0.23
4000	0.18	0.17	0.17	0.18
5000	0.18	0.17	0.17	0.18

Figure A.4: Mapping accuracy of mapping schemes with varying distance, averaged over all KITTI sequences. The overall localization error decreases over longer distances and CarMap's localization error is almost the same as ORB-SLAM2's

Figure A.8 shows the error distribution of CarMap is similar to ORB-SLAM2, and QuickSketch for a map built from, and used in the first KITTI sequence, despite reducing map sizes by a factor of 20.

An important property of a map is to able to localize accurately over long distances. To study how CarMap's localization accuracy changes with the mapped area, we calculate the average translational error at different distances (*i.e.,* 50m to 5km) for all 11 KITTI sequences. We average these errors on all KITTI sequences and report the numbers in Figure A.4. As distance increases, the average translational error decreases and CarMap does as well as ORB-SLAM2 in almost all cases. The reason for this, as mentioned in §3, is that although CarMap removes keyframe-features, the robust feature matching (§3.2) makes up for the 20x fewer features with better matching.

A.5 Map Reconstruction

CarMap reduces map size by trading off compute for storage. The map load time for CarMap consists of the time to load the map from disk and the reconstruction time. After loading the map into memory, CarMap reconstructs two indices and infers the 2D and 3D position of map-features in keyframes (§3.5). Even so, as shown in Figure 20, except for sequence 00, 01 and 06, the load times for CarMap *are less than the ORB-SLAM2 baseline* (on average, 0.95×).

Figure A.5 shows the breakdown of the various map elements that contribute to map reconstruction time for all 11 KITTI sequences. In all sequences, reconstructing the feature-index takes around 40% of the overall reconstruction time. This, however, is still 2-4x less than the reconstruction time for keyframes that contain keyframe-features (in other mapping schemes) instead of just map-features. Calculating the 2D and 3D positions of map-features also takes an average 35% of the overall reconstruction time. The main reason for higher load times (Figure 20), as compared to ORB-SLAM2, in some cases (sequence 00, 01, and 06) is because of the variability map-feature index (orange bar) reconstruction times. The map-feature index is a graph that relates map-points to keyframes they were detected in. Hence, for environments like highways where the scene stays relatively constant, this graph is denser and so the reconstruction costs for the map-feature index are relatively greater. On the other hand, for environments where features change quickly *e.g.,* narrow streets, the map-feature index reconstruction times are lower because these graphs are not as dense. For instance, the feature-index reconstruction for sequence 00 (captured in narrow-streets) is approximately 3x greater than sequence 01 (captured on the highway).

A.6 Map Stitching Evaluation

In this section, we evaluate the ability of CarMap to accurately stitch map segments collected from different spatial and temporal conditions. We compare CarMap against two other map stitching schemes: progressive relocalization and QuickSketch. In progressive relocalization, as opposed to CarMap (one-shot stitching), we relocalize *every* keyframe from the incoming map segment instead of using the global transformation matrix. QuickSketch can only stitch a stereo camera trace with a QuickSketch generated map segment. So, for stitching, QuickSketch loads the QuickSketch map as a base map and then stitches by localizing the stereo camera trace in it.

We evaluate two metrics for stitching: mapping error, and stitching time. After stitching two map segments, we localize a trace in the stitched map and calculate the absolute translational error (m) for each frame. Mapping error is the mean of the translational errors over the whole trace. The stitching time is the amount of time required to do the whole stitch operation of two map segments.

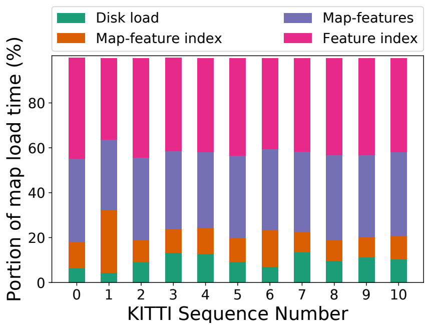


Figure A.5: Breakdown of reconstruction time for CarMap across all KITTI sequences

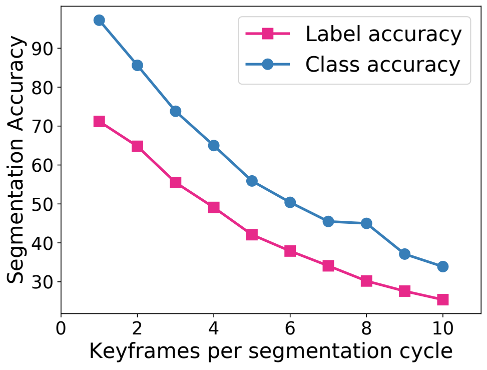


Figure A.6: Semantic segmentation accuracy at different frame rates. If CarMap segments every other keyframe, classification accuracy is 85%.

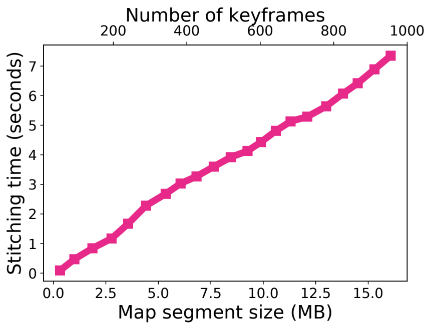


Figure A.7: Computational overhead of stitching. Even map segments as large as 1000 keyframes can be stitched in under 7 seconds.

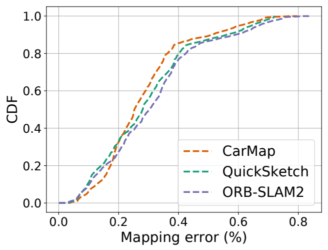


Figure A.8: For a map built, and used from a real-world trace (KITTI Trace 00) 80% of CarMap's mapping errors are less than 0.4% with respect to the length of the trace.

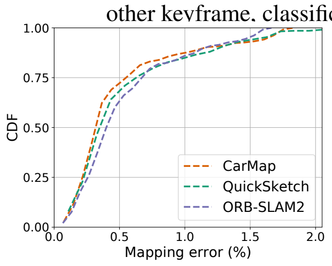


Figure A.9: For a map built, and used in a static trace collected from CarLA, 75% of the mapping errors for CarMap are less than 0.2% with respect to the length of the trace.

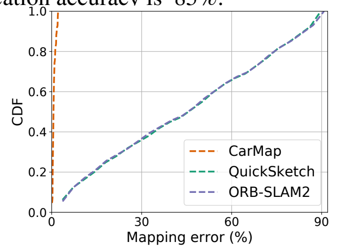


Figure A.10: For maps built, and used in CarLA's dynamic environments, CarMap has a maximum error of 2%. ORB-SLAM2 and QuickSketch have maximum errors of 90%.

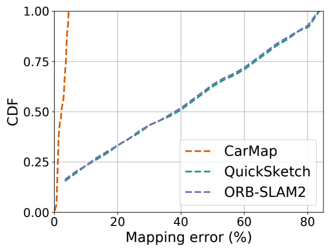
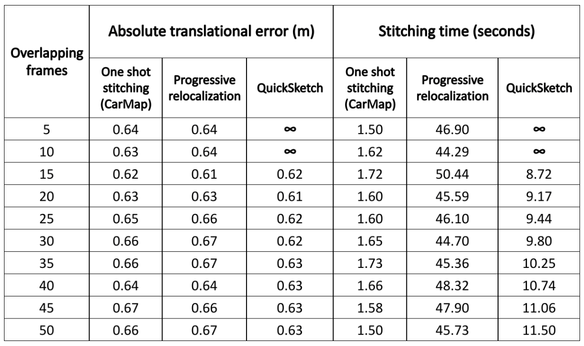


Figure A.11: For CarLA maps built from static, and used in dynamic environments, CarMap has a max error of 4%. ORB-SLAM2 and QuickSketch have maximum errors of 90%.



Overlapping frames	Absolute translational error (m)			Stitching time (seconds)		
	One shot stitching (CarMap)	Progressive relocalization	QuickSketch	One shot stitching (CarMap)	Progressive relocalization	QuickSketch
5	0.64	0.64	∞	1.50	46.90	∞
10	0.63	0.64	∞	1.62	44.29	∞
15	0.62	0.61	0.62	1.72	50.44	8.72
20	0.63	0.63	0.61	1.60	45.59	9.17
25	0.65	0.66	0.62	1.60	46.10	9.44
30	0.66	0.67	0.62	1.65	44.70	9.80
35	0.66	0.67	0.63	1.73	45.36	10.25
40	0.64	0.64	0.63	1.66	48.32	10.74
45	0.67	0.66	0.63	1.58	47.90	11.06
50	0.66	0.67	0.63	1.50	45.73	11.50

Figure A.12: Mapping error (m) with different overlapping regions. CarMap can stitch with fewer overlapping frames than QuickSketch and 30x faster than progressive relocalization.

Stitching Overlap. In the first experiment, we evaluate the mapping error and stitching time of the three mapping schemes as a function of the overlap between the two map segments. For this, we take a single stereo camera trace and split it into two traces with different overlaps. Figure A.12 shows that QuickSketch fails to stitch when the number of overlapping frames between the two map segments is less than 10 frames (1 second). This is because it is not able to find enough feature matches between the two map segments. On the other hand, CarMap can find enough feature matches even though it uses 20x fewer features due to its robust feature matching (§3.2). The mapping accuracy remains relatively constant irrespective of the amount of overlap because CarMap only needs to localize a single keyframe in the base map for a successful stitch operation. Although the mapping error of progressive relocalization is identical to CarMap, it takes approximately 30x more time to stitch the same area. In the stitch operation, localizing a keyframe in the base map is the most expensive operation. CarMap intelligently localizes a single keyframe in the base map and then uses a transformation matrix to shift the remaining map elements. On the other hand, progressive relocalization localizes all keyframes in the base map and hence takes a much longer time. So, as the size of the incoming map segment increases, the stitching time for progressive relocalization will increase significantly.

Stitching Overhead. To study the overhead of stitching, we take a KITTI trace and split it into two map segments (with a few overlapping frames). In doing so, we mark one as the base map and the other as the incoming map segment. We keep the size of the base map constant and vary the size of the incoming map segment. Figure A.7 shows that the stitching time increases with the size of the incoming map segment. It also shows that for map segments containing as many as 1,000 keyframes (15 MB), stitching takes only 7 seconds.

A.7 Semantic Segmentation

In this experiment, we evaluate the object label and class (static, and dynamic) estimation accuracy of CarMap against the frame rate of semantic segmentation. For this experiment, we generate stereo camera traces from CarLA. We segment these images with MobileNetV2. For ground truth, we use CarLA's own semantic segmented images.

Figure A.6 plots the accuracy of segmentation in CarMap using majority voting at different frame rates. We start by running segmentation every keyframe and evaluate till running segmentation every 10 keyframes. In the KITTI dataset, the average keyframes inserted per second is 3.17 and the worst case is 10 keyframes per second. The worse case corresponds to running segmentation every 2 keyframes *i.e.,* a class accuracy of 86% with CarMap using MobileNetv2 in a majority voting scheme.

Food and Liquid Sensing in Practical Environments using RFIDs

Unsoo Ha, Junshan Leng, Alaa Khaddaj, Fadel Adib
Massachusetts Institute of Technology

Abstract – We present the design and implementation of RF-EATS, a system that can sense food and liquids *in closed containers without opening them or requiring any contact with their contents*. RF-EATS uses passive backscatter tags (e.g., RFIDs) placed on a container, and leverages near-field coupling between a tag's antenna and the container contents to sense them noninvasively.

In contrast to prior proposals that are invasive or require strict measurement conditions, RF-EATS is noninvasive and does not require any calibration; it can robustly identify contents in practical indoor environments and *generalize to unseen environments*. These capabilities are made possible by a learning framework that adapts recent advances in variational inference to the RF sensing problem. The framework introduces an RF kernel and incorporates a transfer model that together allow it to generalize to new contents in a sample-efficient manner, enabling users to extend it to new inference tasks using a small number of measurements.

We built a prototype of RF-EATS and tested it in seven different applications including identifying fake medicine, adulterated baby formula, and counterfeit beauty products. Our results demonstrate that RF-EATS can achieve over 90% classification accuracy in scenarios where state-of-the-art RFID sensing systems cannot perform better than a random guess.

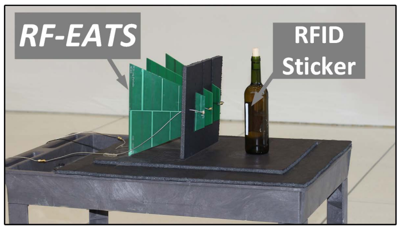
1 Introduction

The networking community has recently witnessed a surge in research that uses wireless signals for sensing liquid and food properties [25, 21, 66, 77]. This research is motivated by a desire to develop low-cost, ubiquitous solutions for food safety sensing by leveraging pervasive networking technologies. In contrast to traditional food sensing solutions which rely on expensive equipment in specialized labs, these new proposals aim to make food safety sensing accessible to lay consumers. This can help avoid widescale future health hazards like the Chinese baby milk scandal [43], the Flint water crisis [70], and the recurring alcohol poisoning problem which results in hundreds of cases of blindness and death every year [32].

Despite initial steps made toward this vision [21, 77, 66], existing proposals still have fundamental limitations that make them too invasive and/or impractical for lay consumers. Specifically, they either require users to extract liquid samples and place them in specialized containers (which often involves a complex calibration process) [21, 77, 56], or they can only operate correctly a single lab setup under strict measurement conditions [66, 25]. These limitations make it difficult for consumers to use such systems for testing products for contamination or counterfeiting before purchasing and outside pre-calibrated lab environments.

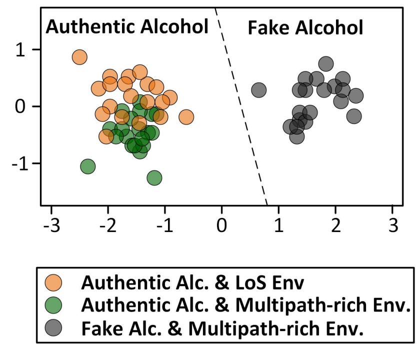
The goal of this paper is to develop a *noninvasive, zero-calibration* system for wireless sensing of food and liquids in practical environments. Such a system would enable consumers to test food and liquids without opening their containers and in different environments: supermarkets, grocery stores, or homes. Our system will rely on off-the-shelf RFIDs (Radio Frequency IDentifiers), similar to those used in some past proposals [25, 66]. RFIDs cost few cents each, and they have been widely adopted by the industry as barcode replacements for billions of items (including food products). This makes them ideal candidates for low-cost and ubiquitous food sensing. Moreover, our recent research has demonstrated that an RFID's signal changes when it is placed on containers filled with different liquids due to near-field coupling between the RFID's antenna and material inside the container [25]. However, similar to earlier wireless proposals, this RFID-based approach could not generalize to new environments.

The difficulty in extending wireless food sensing to different environments is that radio signals are not only impacted by the content of a container but also by the environment where the measurement is made. Fig. 1 illustrates this challenge by showing three experimental trials in two different setups. Each setup consists of a wireless reader that measures the RFID's signal and extracts the channel response. This response is impacted by two factors: the content inside the container (due to the near-field coupling) and the measurement environment which encompasses the location of the container with respect to the reader as well as the reflections off different objects in the environment (due to the propagation of the RFID's wireless signal before it reaches the reader). As a result, if either the environment or the content proper-

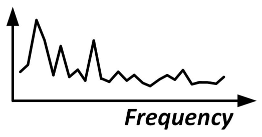


(a) First Environment (E1)

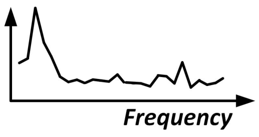
(b) Second Environment (E2)



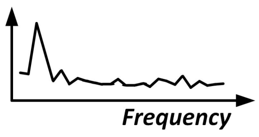
(f) Learning Results



(c) Authentic in E1



(d) Authentic in E2



(e) Fake in E2

Figure 1: **RF-EATS senses food and liquids noninvasively across different environments.** The figure shows how RF-EATS uses unmodified RFID stickers placed on authentic and fake alcohol-filled bottles to sense their contents noninvasively in two different environments shown in (a) and (b). (c)-(e) plot the RFID's wideband response from three experimental trials. Because the environment significantly impacts the measured RFID response, the frequency spectra of (d) and (e) look closer than (c) despite that (c) and (d) are from authentic alcohol bottles while (e) is from a fake alcohol one. (f) plots the classification output of RF-EATS's learning model (after dimensionality reduction), where each point represents a different experimental trial. The plot shows that trials with authentic alcohol collected across different environments are clustered together, while those with fake alcohol are in a different cluster. This shows that RF-EATS can correctly classify contents despite the environmental changes.

ties change, the response changes (see the bottom row of Fig. 1). This is why prior proposals require calibrating or constraining the measurement conditions, limiting the practicality and generalizability of their designs.

We present RF-EATS,[1] an RFID-based sensing system that is robust to environmental variations and that generalizes well to unseen environments. At the core of RF-EATS's design is a neural learning model that can learn RF features due to a container's content and discard those resulting from extraneous environmental changes. For example, if the system is trained to detect adulterated baby formula in a lab environment, it still has high detection accuracy in a supermarket-style setup with dense environmental multipath (reflections) from metal shelving and other items on the aisles.

A fundamental challenge in training any neural learning model (including for image or text classification tasks) arises from the need for very large datasets. This challenge is exacerbated in our context due to the limited availability of RF datasets for sensing. A naive solution is to collect an extensive dataset that covers various indoor environments and use it in training. However, such an approach is time-consuming, inefficient, and incapable of generalizing to unseen environments.

To efficiently generalize to different environments, RF-EATS builds on recent advances in variational autoencoders [53, 33] and adapts them to RF sensing tasks. These models are typically used to generate realistic synthetic data (e.g., images of faces of humans who do not exist). Instead, RF-EATS employs them to generate a large number of realistic multipath-affected data from a small number of real-world measurements. To do so, it introduces a *multipath kernel function*, which allows it to (approximately) decompose the wireless channel into content-dependent and environment-dependent features. Subsequently, RF-EATS can train an autoencoder to learn distributions of practical radio environments (i.e., reflections, position changes, etc.) by focusing on the environment-dependent features. This allows RF-EATS to emulate a large number of realistic measurements and use them to train its neural classifier.[2] In §3.1, we describe this technique in detail and demonstrate how its stochastic nature enables generalizing RF-EATS to unseen environments.

It is desirable to extend RF-EATS's learning framework to new contents in a sample-efficient manner. For example, if a model is trained to detect adulterated baby formula, we would like to extend it to detect fake alcohol using a small number of alcohol measurements. Said differently, we would like to harness the power of a well-trained model on a large number of measurements to achieve high accuracy on new tasks, without having to train a new model from scratch. To do so, RF-EATS employs transfer learning: it divides a multi-layered network into *common layers* (shared by all tasks) and *task-specific layers*. In order to learn a new task (e.g., detecting fake alcohol), it can inherit the common layers from a well-trained model (e.g., the baby formula model) and only needs to retrain the task-specific layers. This further reduces the number samples required to extend the model to new contents, allowing RF-EATS to achieve near-

[1]RF-EATS stands for RF-based Environment-Agnostic Transferable Sensing.

[2]In our evaluation, we demonstrate how this approach significantly outperforms using the standard ray-tracing model for generating synthetic multi-path environments [63].

optimal accuracy even when the dataset from a new content is limited. We describe this model in details in §3.2.1 and show how the common layers can serve as a pre-trained model for future classification tasks beyond those described in this paper.

We implemented a prototype of RF-EATS on USRP X310 software radios, and tested it with off-the-shelf UHF (Ultra-High Frequency) RFIDs. We adapted a recent wideband measurement technique which can extract more than half a GHz of RF measurements from off-the-shelf, passive RFIDs [40], thus providing a rich set of features for classification. We evaluated it in seven different applications and with sixteen different contents including: fake medicine, adulterated baby formula, contaminated alcohol, counterfeit perfume, wine aging, and soda classification. In each of these applications, we evaluated its ability to identify adulteration under standard contamination/adulteration levels reported in recent cases [43, 32, 50, 17, 19].

Our results from 2048 experimental trials in 20 different environments demonstrate the following:

- RF-EATS's accuracy approaches 90% across most the above applications even when tested in new and unseen environments. In contrast, the accuracy of a state-of-the-art baseline [25] drops from 90%+ when evaluated in the same environment to a random guess in unseen environments for half the applications.
- RF-EATS's transfer model enables achieving near-optimal accuracy with as little as four data samples, demonstrating the importance of this model when the dataset is limited.
- We show how RF-EATS's autoencoder can be used as an anomaly detector to generalize to contaminated or counterfeit content even if it has not been trained on the specific contaminant.
- RF-EATS's accuracy is directly impacted by the dielectric differences between the contents it wishes to classify. We show some negative results including its low accuracy in detecting fake extra-virgin olive oil due to limited dielectric differences between fake and authentic olive oil.

Contributions: RF-EATS is the first RFID-based system that can noninvasively sense food and liquids in closed containers and operate correctly in unseen environments. It employs a variational autoencoder architecture that can learn and generate realistic multipath environments, and introduces a new kernel function that can apply these generated environments to real data. It also employs transfer learning to efficiently extend its sensing capabilities to new liquids and food items. The paper also contributes a prototype implementation and evaluation of RF-EATS in practical environments.

It is important to note that RF-EATS's performance is directly impacted by the extent of dielectric differences between contents it wishes to classify. This means that if the dielectric differences are small (e.g., the olive oil application), the accuracy degrades. This degradation is likely to be mitigated as the dataset and learning models evolve. We also note that our evaluation focused on demonstrating robustness to changes in the surrounding environment while fixing the container's material (e.g., glass or plastic) and shape. Despite these limitations, RF-EATS marks an important step toward food and liquid sensing in practical environments. More generally, we hope that RF-based liquid and food sensing will follow a similar trend in accuracy improvements as that witnessed by vision and text learning tasks in recent years.

2 Background

Researchers have long recognized the need for monitoring food quality and safety. Most existing techniques rely on measuring electrochemical and electrophysical properties [42, 35, 65, 1]. The process involves extracting food samples and placing them in direct contact with chemical reagents and/or specialized sensing circuits (e.g. biotoxin sensors [15, 16, 27]) and is typically done in specialized food labs.

Given the length of the food lab testing process, recent proposals have considered building small sensing circuits in hope of incorporating them inside food containers [72, 46, 62, 51, 37]. These proposals require designing a customized sensor for every different type of food or food property of interest [72, 62] or they require coating existing circuits (e.g., LC circuits or RFIDs) with different types of polymers to increase sensitivity to specific materials of interest [46, 51]. Moreover, many of these sensors still require direct contact with food samples, which can lead to contaminating the food samples and is erosive to their sensing interfaces [47, 76, 58].

The desire for ubiquitous and general purpose solutions has led networking and mobile researchers to explore various mobile sensing modalities. These techniques rely on different kinds of wireless signals to extract material properties such as the electric permittivity [66, 21, 23, 44, 78], surface tension [60, 71, 77], or photo-acoustic signatures [57]. However, the reliance on wireless signals makes these techniques highly sensitive to measurement conditions; hence, the proposed systems require isolating food samples of interest and placing them in calibrated setups. This includes recent proposals like LiquID [21], TagScan [66], and CapCam [77]. The invasiveness of these approaches makes them unsuitable for use by consumers before they purchase counterfeit or contaminated food and liquid products. RF-EATS shares the vision of this line of work but aims to develop a non-invasive approach for food and liquid sensing.

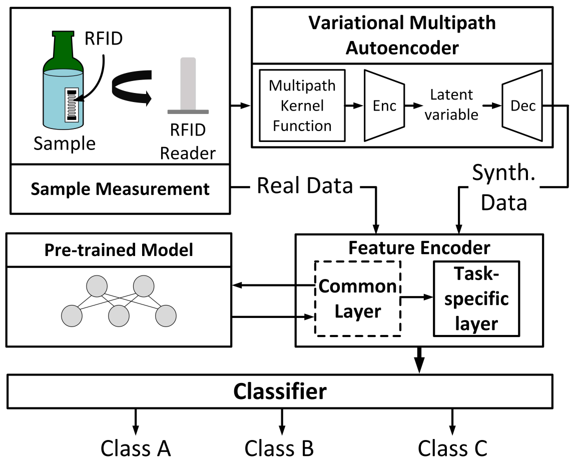


Figure 2: **RF-EATS's Learning Model.**

3 Design

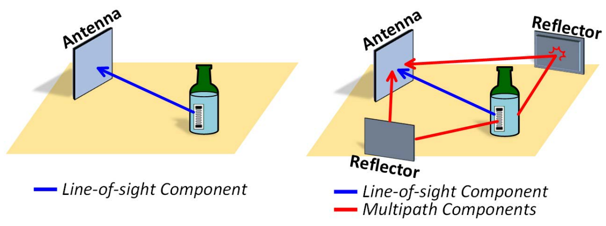
RF-EATS is a wireless system that can noninvasively sense food and liquid products in closed containers without opening them. It relies on cheap, passive UHF (Ultra-High Frequency) RFID tags placed on the containers, and exploits the near-field coupling between the RFID's antenna and the container's contents. These RFIDs may either be already affixed on the container by the manufacturer, or they may be attached as stickers to the container as shown in Fig. 1.

Fig. 2 shows the overall system architecture. From the hardware perspective (top left), it uses a modified RFID reader capable of obtaining wideband channel measurements [39, 25, 14] of unmodified RFID tags. The reader sends a downlink signal which powers up a passive tag and obtains the tag's response. It performs standard channel estimation using the packet preamble, and uses the channel estimates in training and classification.[3] Note that while our discussion focuses on a single tag, it can be easily extended to any number of tags since it adapts the standard EPC-Gen2 protocol.

RF-EATS can be used for various inference tasks (e.g., detecting adulterated baby formula, fake alcohol). Its learning architecture consists of a neural network with the following components, as shown in Fig. 2:

- *Variational Multipath Autoencoder (§3.1):* This component takes as input a small number real-world channel measurements and outputs a large number of realistic synthetic measurements. At the heart of this component is a multipath kernel function that enables learning representative distributions of RF environments while discarding the container contents' impact.
- *Feature Encoder (§3.2.1:)* This component takes as input real and synthetic data and extracts features for use in classification. It consists of multiple layers, some of which are shared among all classification tasks. These layers may be reused as a pre-trained model for extending RF-EATS to new types of contents.
- *Classifier (§4.1):* The component takes as input the features outputted by the feature encoder, and outputs the classification results. While the variational autoencoder and part of the feature encoder are shared by all tasks, this component must be retrained for each task.

[3]It relies on the *channels not the IDs* for the inference tasks.



(a) Line-of-sight environment (b) Multipath environment

Figure 3: **Typical Ray-Tracing Channel Approximations.**

The above three components together enable RF-EATS to generalize to unseen environments (through synthetic data from the autoencoder), expand to new materials using small datasets (by leveraging the shared layers of the feature encoder), and extend to untested contaminants (by using the autoencoder as an anomaly detector). The following sections explain these components in details.

3.1 Variational Multipath Autoencoder

To achieve high accuracy with a neural learning model, RF-EATS's training dataset needs to be large and representative of a variety of environments. Unfortunately, collecting large datasets for every contaminant and every multipath environment is an expensive and time-consuming [28]. Further, even if we manage to collect such datasets, there would remain unseen environments which the model may not be able to generalize to.

Below, we describe how RF-EATS overcomes this challenge by leveraging a stochastic generative model based on variational autoencoders. The model enables it to generate realistic synthetic data for use in training, which increases its accuracy despite limited datasets and enables it to generalize to unseen environments.

3.1.1 The Multipath Kernel

RF-EATS's generative model needs to realistically capture different aspects of an RFID's measured channel response. Fig. 3(a)-(b) depict common approximations of the wireless channel. In line-of-sight scenarios, the RFID's wireless signal arrives on a direct path to the reader's antenna; in multipath-rich environments, the signal arrives on multiple paths (after bouncing off various reflectors) which linearly combine at the receiver. Mathematically, the RFID's channel h at a given frequency k is typically approximated as [63]:

$$h_k = \sum_{i=0}^{N} a_i e^{-j2\pi f_k \tau_i} \tag{1}$$

where a_i and τ_i are the amplitude and time delay of the i^{th} path, f_k is the k^{th} frequency, and N is the total number of paths.

This standard approximation is problematic for RF-EATS's learning tasks because of two main reasons: 1) it ignores the impact of the RFID's antenna gain and 2) it ignores scattering and diffraction phenomena of radio signals. The first approximation is particularly detrimental since it prevents capturing the impact of the container's content on the antenna (more specifically, the impact of the content's dielectric ε). The second one is also problematic as it results in less representative channel distributions.[4] While it is possible to overcome these shortcomings by solving Maxwell's equations [64], this is undesirable since it requires precise modeling of the geometry and materials in the environment, making it practically and computationally expensive [61].

To truthfully represent the RFID's measured response, we would like RF-EATS's generative model to embrace the complexity of the wireless channel. The model must incorporate both the impact of the content dielectric (on the antenna gain) and that of the wireless signal propagation (due to reflection, scattering, and diffraction). We can achieve this by expressing the overall channel as a product of the gain $G(\varepsilon, k)$ and the propagation $P(k)$ characteristics as follows:

$$h_k = G(\varepsilon, k) \cdot P(k) \tag{2}$$

We make the following remarks:

- First, one might wonder why the presence of other nearby objects does not impact the RFID's antenna gain (i.e., why it only affects the propagation factor $P(k)$). To answer this question, we note that the electromagnetic interaction of antennas with different objects in the environment depends on the distance between the antenna and the objects [29]. If an object is in the near-field (i.e., within one wavelength[5]), it "couples" with the antenna and impacts its gain. If the object is in the far-field (i.e., larger than two wavelengths), it impacts the propagation $P(k)$. This is why RF-EATS incorporates the impact of container contents into the gain while absorbing environmental multipath into the propagation factor.[6]
- If the location and multipath environment are fixed, then any change in the measured channel h_k can be attributed to the gain $G(\varepsilon, k)$ and thus be used directly to infer the contents. Indeed, this is why past proposals for wireless food sensing required fixing and/or calibrating their setup but are unable to generalize to different multipath environments.
- Similarly, if the content of a container is fixed, then any change in the measured channel can be attributed entirely to the multipath environment $P(k)$. Let us say that we measure the RFID's channel in two scenarios: one in a line-of-sight (LOS) controlled environment with little to no multipath ($h_{k,LOS}$), and another in a multipath-rich environment ($h_{k,MPATH}$). The ratio of these measurements is entirely dependent on the multipath environment and independent of a container's contents. Specifically:

$$\frac{h_{k,MPATH}}{h_{k,LOS}} = \frac{P_{MPATH}(k)}{P_{LOS}(k)}$$

We call this the *multipath kernel function*. For simplicity, we approximate a multipath-free $P_{LOS}(k) \approx 1$.

[4]In §4.4, we empirically compare against the standard ray-tracing model and show that RF-EATS significantly outperforms it.

[5]In the UHF ISM band, the wavelength is about 30 cm. It becomes significantly smaller in liquids due to the impact of the dielectric.

[6]This approximation works well in practical because near-field backscatter power decays as $1/d^4$.

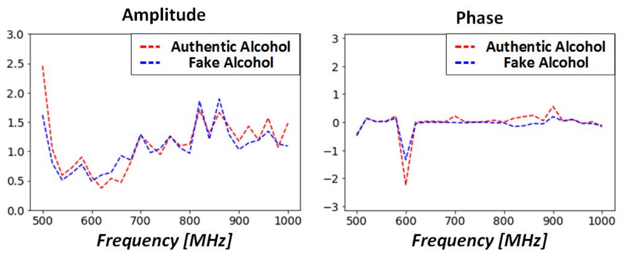


Figure 4: **Output of Multipath Kernel Function.**

To test that the multipath kernel indeed results in content-independent features, we ran experimental trials with fake and authentic alcohol in two environments representing line-of-sight and multipath-rich settings similar to those shown in Fig. 1. We computed the ratio of the channel in multipath to that in line of sight, and plot the output in Fig. 4. Since the channel is a complex number, we plot the amplitude and phase of the ratios on separate graphs, each as a function of frequency. The figure shows that the ratio is indeed independent of the content since the plots for authentic (red) and fake (blue) alcohol almost overlap for both magnitude and phase. This indicates that the kernel function enables us to extract environment-dependent content-independent features.

3.1.2 Training a Generative Model

Now that we have a mechanism to obtain environment-dependent features from real-world measurements, we can use them to train a generative model. The model takes these measurements as input and generates a large number of synthetic data representing different multipath environments. While there are various kinds of generative models [55, 53, 33], RF-EATS employs variational autoencoders (VAE) because they have the ability to generalize using a small input dataset. In what follows, we

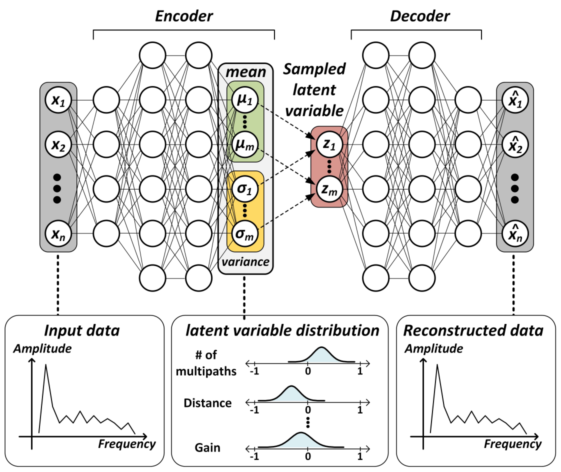


Figure 5: **RF-EATS's Variational Autoencoder.**

describe these models at a high level and in the context of our problem, and we refer the interested reader to [53] for a more detailed exposition.

VAEs assume that the input features represent a much lower dimensional space of latent variables. In the context of our learning tasks, the wireless propagation factor is indeed caused by a small number of reflectors and scatterers in the environment [34]. However, the reflectors and scatterers change across different environments, resulting in different channel responses. VAEs capture this phenomenon by assuming that the latent variables are randomly drawn from a normal distribution. Once the underlying distributions are learned, RF-EATS can draw new samples from them to generate synthetic data for unseen multipath environments.

Fig. 5 shows the overall architecture of RF-EATS's variational autoencoder. The VAE takes as input the channel ratios, each of which consists of a multi-dimensional vector $x_1, x_2, ..., x_n$, and outputs a reconstruction of these features $\hat{x}_1, \hat{x}_2, ..., \hat{x}_n$. The model consists of an encoder (which aims to compress these features into latent variables $z_1, z_2, ..., z_m$) and a decoder (which aims to reconstruct the input from the latent variables). By compressing and decompressing the input, the neural network aims to learn a representative lower-dimensional distribution of the latent variables.

Formally, the purpose of training the VAE is to learn the parameters of the neural network that can (1) minimize the reconstruction loss between the input $\mathbf{x}$ and output $\hat{\mathbf{x}}$, and (2) model the underlying distribution of the latent variables $q_\theta(z|x)$ as a normal distribution. Formally, this can be achieved by minimizing the following loss function [33]:

$$L_i(\phi, \theta, x_i) = KL(q_\theta(z|x_i)||\mathcal{N}(0,1)) + \mathcal{L}(x, \hat{x}) \quad (3)$$

where KL is Kullback-Leibler divergence, which is a measure of the difference between two probability distributions, $\mathcal{N}(0,1)$ denotes the Gaussian with zero mean and standard deviation of 1, and $\mathcal{L}$ represents the *L2 norm* of the reconstruction loss in frequency domain.

The following points are worth noting:

- The VAE input and output are independent of the content. Hence, it can be trained on any container content (or even on empty containers), and its output may be used for any classification task as we explain below.
- Our discussion focuses entirely on the RFID's channel and ignores the communication bits. This is possible because RF-EATS applies standard channel estimation on the RFID packet's preamble to extract the channel. It also uses an out-of-band sensing technique (described in §4.1) to obtain wideband estimates.

3.1.3 Embedding Dielectric Characteristics

Once the VAE has been trained, it can be used to generate synthetic multipath environments by randomly drawing samples z from the latent distributions and passing them through the decoder shown in Fig. 5. Note, however, that the VAE's synthetic output cannot be directly used to train a contamination classifier (i.e., we cannot directly use it to train a fake alcohol classifier). This is because output features are independent of the content.

In order to generate synthetic measurements that incorporate the impact of both the content and the propagation environment, we need to apply the inverse of the multipath kernel. Specifically, we need to measure $h_{k,LOS}$ of an RFID placed on fake and authentic alcohol bottles in line-of-sight settings, then multiply these measurements by the output features of the VAE. Since the generative model is capable of stochastically generating different multipath environments, we only need a small measurement dataset of $h_{k,LoS}$ to generate a large number of realistic channel measurements and feed them into the classifier. Hence, the VAE model provides a large corpus for training the classifier without requiring measurements for every multipath environment and contaminant.

3.2 Extending to New Tasks and Compositions

In this section, we describe how we can efficiently extend RF-EATS's learning framework to new tasks and unseen compositions or contaminants.

3.2.1 Transfer Learning to New Tasks

We would like to extend RF-EATS to new classification tasks in a sample-efficient manner. For example, having trained a classifier to detect fake alcohol using a large dataset, we would like to extend it to detect adulterated baby formula using a small number of samples. This would enable expanding RF-EATS to new tasks using a smaller number of measurements of the new content of interest. To do so, RF-EATS employs transfer learning in order to transfer training knowledge from a well-trained source domain (e.g., alcohol) to a new target domain (e.g., baby formula).

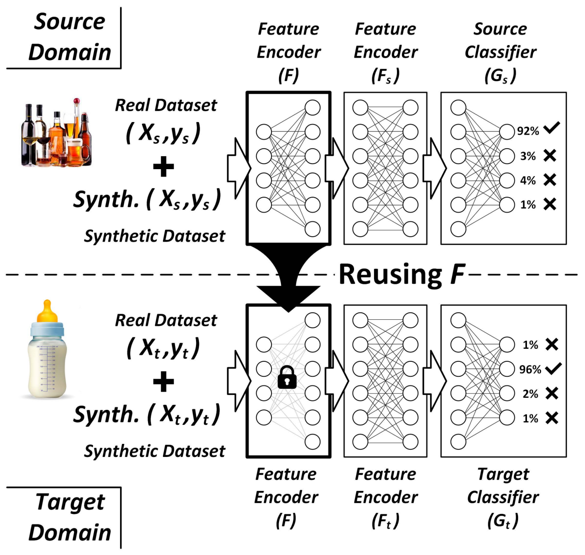


Figure 6: **RF-EATS's Transfer Learning Model.**

Fig 6 shows how this process works at a high level. Recall that neural networks consist of multiple layers, some of which are used for feature extraction or encoding, while others are used for classification. The feature extraction layers themselves may be further divided into common layers and task-specific layers. The common layers can be directly transferred as "frozen" layers F from a well-trained classifier to the target domain. This significantly reduces the number of parameters that need to be learned for the new task, thus reducing the required dataset size to achieve high accuracy.

Mathematically, each domain can be represented as

$$\begin{aligned}(X_s, y_s) &= \{(x_{s,i}, y_{s,i})\}_{i=1}^{N_s} \\ (X_t, y_t) &= \{(x_{t,i}, y_{t,i})\}_{i=1}^{N_t}\end{aligned} \tag{4}$$

where X_s and y_s are the observations and respective labels of the source content domain, X_t and y_s are the observations and labels of the target content domain, x_i represents the complex channel estimate of one observation over different frequencies, y_i represents the label (*e.g.* contaminated or not), N_s the number of samples in the source content domain and N_t the number of samples in the target content domain.

Assuming the latent space dimensionality is d, we can represent the feature encoder and source/target classifier as:

$$\begin{aligned}F &: \mathbb{R}^n \to \mathbb{R}^d \\ G_s \circ F_s &: \mathbb{R}^d \to \{0, 1, \ldots, C_s\} \\ G_t \circ F_t &: \mathbb{R}^d \to \{0, 1, \ldots, C_t\}\end{aligned} \tag{5}$$

where n comes from the dimension of the input data, and C_s, C_t represent the number of classes in the source and target content domains respectively.

After training the model on the source content domain, we fix the weights of F. We then compute $\hat{y}_t = G_t \circ F_2 \circ F(X_t)$ and minimize the standard cross-entropy loss function $\mathcal{L}_t(y_t, \hat{y}_t) = -\sum_{i=1}^{C} p(y_{t,i}) \log p(\hat{y}_{t,i})$, where $y_{t,i}$ and $\hat{y}_{t,i}$ represent the actual and predicted label for class i. Here, only the weights of F_2 and G_t are updated, while the weights of F remain intact.

Algorithm 3.1 VAE-based Anomaly Sample Detection

```
Input: Anomalous data point $x_i$, threshold $\alpha$, trained
  Encoder $E_\theta$ and Decoder $D_\phi$
Output: Reconstruction probability $P_R$
  $\mu_{z(i)}, \sigma_{z(i)} = E_\theta(x_i)$
  draw $L$ samples from $z \sim N(\mu_{z(i)}, \sigma_{z(i)})$
  for $l = 1$ to L do
    $\hat{x}_{(i,l)} = D_\phi(z_{(i,l)})$
  $P_{R(i)} = \frac{1}{L} \sum_{l=1}^{L} \mathcal{L}(x, \hat{x}_{(i,l)})$
  if $P_{R(i)} > \alpha$ then
    $x_i$ is an anomaly
  else
    $x_i$ is not an anomaly
```

3.2.2 Anomaly Detection for Unseen Compositions

So far, we have assumed that RF-EATS's classifiers have been trained on samples from all classes of interest, including counterfeit and adulterated samples. However, in many practical applications, we may not have access to counterfeit or contaminated samples, or the composition/type of contaminant may be unknown.

We can generalize RF-EATS to deal with such situations by using its VAE as an anomaly detector [13, 41]. Recall that the VAE is trained to minimize the reconstruction loss of the environment-dependent features. Hence, if the input to the VAE encodes environment-dependent features, we expect the reconstruction loss to be low. On the other hand, if the input deviates from the expected distribution, the reconstruction loss will be high, indicating an anomaly.

To see how this can be used to detect counterfeiting, consider the case of a manufacturer that creates a database of $h_{k,LOS}$ measurements of the authentic product. Subsequently, if we measure the channel $h_{k,c}$ of a counterfeit product and apply the multipath kernel to it, we obtain:

$$\frac{h_{k,MPATH}}{h_{k,LOS}} = \frac{P_{MPATH}(k)}{P_{LOS}(k)} \times \frac{G(\varepsilon', k)}{G(\varepsilon, k)}$$

Notice how in such situations, the impact of the content does not cancel out upon applying the multipath kernel, and the resulting ratio is not only dependent on the environment P, but also on the content. Thus, if this ratio is fed as input to the VAE, we expect a high reconstruction loss since the sample deviates from the learned distribution. In contrast, if the sample were authentic, the ratio will be dielectric-free and the reconstruction loss would be lower. Algorithm 3.1 summarizes the anomaly detection algorithm using the variational autoencoder.

4 Implementation and Evaluation

4.1 Implementation

Hardware. We implemented our design on USRP X310 and N210 software radios [11] by extending a two-frequency excitation prototype [40]. The radios run the EPC-Gen2 protocol and transmit two frequencies: one high power frequency (10-31dBm) inside the UHF ISM band and another low power sensing frequency which is varied within 500-1000 MHz. At the sensing frequency's receiver, we employed a low-pass filter that eliminates the impact of the power up frequency and added an LNA to boost the received signal power. To reduce the harmonics of transmission, we used a low-pass filter at the output of the transmit USRPs. We also added anti-aliasing low-pass filters at the input of the receive USRPs. The received signal is sampled (digitized) and sent over Ethernet to a computer for offline processing.

Software. RF-EATS's software package was implemented in MATLAB and python. The transmitted query requests the extended RFID preamble, and the receiver averages 50 RFID responses to boost the signal-to-noise ratio (SNR). The receiver decodes the response and performs standard channel estimation using the packet preamble. It repeats this process over 26 frequencies, separated by 20 MHz across RF-EATS's 500 MHz frequency span. This results in RF-EATS's feature vectors which include amplitude, phase, and correlation across frequencies. The multipath kernel was applied by dividing the multipath-affected data by line-of-sight measurement data ($h_k/h_{k,LoS}$). The number of datapoints used in training the classifiers is much larger than the number of measurements due to this combinatorial relationship.

To ensure the reliability of the channel measurements (i.e., that the channel estimates were not significantly distorted by noise or interference), we computed the correlation at each frequency k as:

$$corr_k = \sum_t y_t p_t^* / \sqrt{\sum_t |y_t|^2 \sum_t |p_t|^2} \qquad (6)$$

where p_t is the known preamble of the RFID packet and y_t is the received signal. We discarded points that had very low ($\leq$0.6). We used the python implementation of the PyTorch and Keras package [49] to implement RF-EATS's classifier and refiner.

Transfer Learning Classifier. The classifier was implemented as a fully-connected network with 3 hidden layers. Dropout and batch normalization layers were added to minimize overfitting. We used the Adam optimizer and set learning *rate = 1e-4, beta1=0.9, beta2=0.999, dropout rate=0.2*.

Variational Autoencoder. The encoder and decoder were implemented as fully-connected networks with 3 hidden layers each. The dimension of latent variable was set to 16. We used the Adam optimizer and set learning *rate = 1e-7,beta1=0.9, beta2=0.999, dropout rate=0.2*.

Content	Samples	Content	Samples
Pure Alcohol	218	Baby Formula (uncontaminated)	95
Diluted Alcohol (10% water)	218	Baby Formula (Contaminated)	94
Tainted Alcohol (10% methanol)	218	Extra Virgin Olive Oil (unadulterated)	80
Coke	218	Extra Virgin Olive Oil (adulterated)	79
Pepsi	218	Wine (2009)	77
Diet Coke	116	Wine (2012)	76
Real Perfume	102	Fake Medicine	68
Counterfeit Perfume	103	Real Medicine	68

Table 1: Number of Samples per Material

4.2 Dataset & Applications

We tested RF-EATS in 7 different applications and collected 2,048 data samples in total. The applications demonstrate the generality of the technique to important real-world tasks. Below, we describe these applications, their motivations, and how their corresponding compositions were obtained. The dataset is detailed in Table 1.

- **Tainted Alcohol and Diluted Alcohol.** Tainted alcohol is an ongoing problem in many developing world countries including China, Indonesia, Iran, Turkey, India, and Mexico [32]. Alcohol is tainted by mixing it with cheaper methanol, and consuming it leads to hundreds of cases of blindness and death every year. Standard tainting percentages range between 30-50%. In order to stress-test for sensitivity, we prepared tainted alcohol by removing 10% of the content of an authentic bottle of GRAVES Grain Alcohol [8] and replacing it with methanol.
- **Adulterated Baby Formula.** In 2008, the Chinese milk scandal broke out after the hospitalization of 50,000 babies due to kidney damage [43]. Manufactures had watered down baby formulas up to 83% and mixed them with melamine CAS NO. 108-78-1 [24], a compound used in making plastics. The purpose of adding melamine (by manufacturers) was to conceal dilution by artificially increasing protein levels. To stress-test the sensitivity of our system, we prepared adulterated baby formula by diluting a bottle of Enfamil NeuroPro Infant Formula - Ready to Use (8 fl Oz) [6] with 50% water and mixing it with a higher concentration of melamine (1g/L).
- **Fake Medicine.** Fake medicine is also a major challenge in many developing-world countries, leading to dozens of fatalities every year [50]. A recent incident involved fake cough medicine bottles, where 90% of the active ingredient was replaced with diethylene glycol, a compound used in making antifreeze agents [54]. To prepare such samples, we removed

80% of the contents of a Tylenol bottle [3] and replaced it with diethylene glycol.

- **Fake Extra-Virgin Oil.** Recent studies have shown that 69% of US-imported Extra Virgin Olive Oil has been adulterated by mixing it with cheaper oils (*e.g.* peanut oil) [26]. This can lead to health hazards for consumers with (peanut) allergy. Standard adulteration levels range between 70-80% [26]. We prepared fake olive oil by removing 80% a bottle of GOYA Extra Virgin Olive Oil [7] and replacing it with peanut oil from Planters 100% Pure Peanut Oil.
- **Counterfeit Perfume.** Counterfeit beauty products are abound, leading Estee Lauder to confiscate over 2.6 million counterfeit items in 2016 alone [17]. Many such products are sold online. We purchased an authentic Chanel perfume (COCO MADEMOISELLE - Eau de Parfum) directly from the supplier (160$) and a knock off for $40 from an online retailer.
- **Wine Fraud.** Wine fraud takes many forms. A common one involves selling consumers wine vintages that are dated to earlier years, artificially inflating their price [19]. We purchased wine vintages of Castalia Pinot Noir from two different years: 2009 and 2012.
- **Soda Brand.** Counterfeit soft drinks are marketed under common brand names [4]. While it was difficult for us to purchase counterfeit soft drinks, we tested RF-EATS's ability to classify between common brands: Coke, Pepsi, and Diet Coke.

4.3 Evaluation

Environments & Setup. We evaluated RF-EATS in 20 environments in total including supermarket-style setups with dense metal shelving, kitchens (with sinks and fridges), open lab spaces, offices, hallways and corridors, dining table settings, etc. These environments were fully furnished with tables, chairs, and computers. People walked around during our measurements, and various wireless technologies were present (LTE, WiFi, Bluetooth, etc.). Fig. 1 shows two sample setups, one representing an open lab space and another emulating a supermarket environment.[7] In each experimental trial, a container with an RFID was placed within 10-20 cm distance and -45°-45° from RF-EATS's antennas. The device powers up the RFID and captures its response. Across our trials, we varied the measurement conditions by changing the location of the container and the number, location, and kinds of objects/reflectors around it. Note that even though the RFID is relatively close to RF-EATS's antennas,[8] the richness of the multipath environment significantly impacts the measured response.

[7]Most of our trials (aside from the line-of-sight measurements) were performed outside the clean open space environments.

[8]Beyond such distance, it is difficult to power up an RFID on liquid-filled containers.

RFIDs. We performed our experiments with a variety of commercial off-the-shelf passive UHF RFIDs, including the Alien ALN-9640 Squiggle [2] and Smartrac [59] tags. Each tag costs around 5 cents.

Ground-truth. To measure the ground truth dielectric constant, we used a vector network analyzer, the Agilent Technologies E8362B PNA Network Analyzer [5] (price∼$20,000), and connected it to an N1501A dielectric probe [9]. We measured the dielectric constants across 500-1000 MHz frequency range for each content.

Baselines. Our baseline evaluation focused on non-invasive proposals (i.e., we avoided past systems that are invasive or require isolating liquid samples, e.g., [21, 77]). We implemented the following baselines:

- *RFIQ (Gradient Boosting) [25]:* We implemented RFIQ's gradient boosting tree model. We set $eta = 0.3$, $max\_depth = 3$, $subsample = 0.5$, and $num\_boost\_round = 128$.
- *VAE with a Ray-Tracing Model:* We implemented a VAE that is trained on an analytical ray-tracing model. Specifically, rather than collecting $h_{k,MPATH}/h_{k,LOS}$ in a data-driven fashion, we modeled it using the following equation:

$$\frac{h_{k,MPATH}}{h_{k,LOS}} = \sum_i a_i e^{-j2\pi f_k \tau_i} \tag{7}$$

  where a_i and τ_i are the amplitude and time delay of the i^{th} path, and f_k is the k^{th} frequency. This model allows us to generate a large number of synthetic multipath environments and use them in training.
- *Simple Neural Network:* We also implemented a 3-layer fully-connected neural network, which has the same structure with RF-EATS's neural network (but without the transfer learning and VAE components).

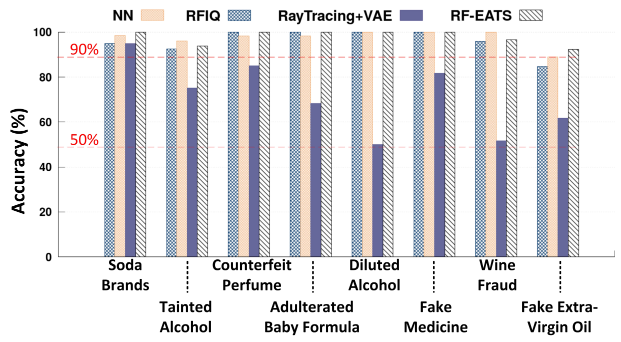
We note the following additional points:

- Across our evaluation, the dataset was divided into a training, testing, and validation sets, all mutually exclusive, i.e., none of the containers or measurements used in one of these sets was present in any other sets.
- We trained the VAE using measurements taken from an RFID placed on empty containers. Recall that the VAE is content-agnostic and general.
- Similar to standard machine learning approaches, we extensively explored the space of hyper-parameters and scaling functions in our training process.

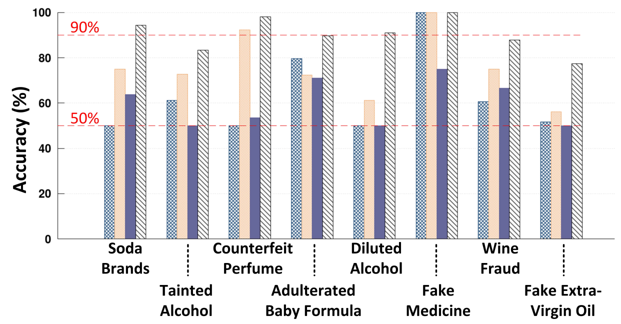
4.4 Results

4.4.1 Overall Performance

We would like to evaluate RF-EATS's overall performance in two different regimes: the first involves training and testing in the same environment, while the second involves training in one environment and evaluating in all



(a) Training and testing in the same environment.



(b) Training in one environment and testing in other environments.

Figure 7: **Overall Performance** (a) and (b) plot the accuracy for each of the applications with different experimental regimes.

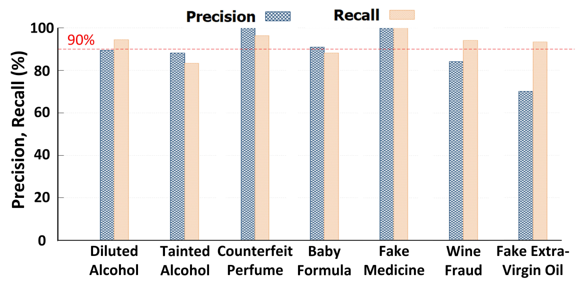


Figure 8: **Precision and Recall of Counterfeit Detection.**

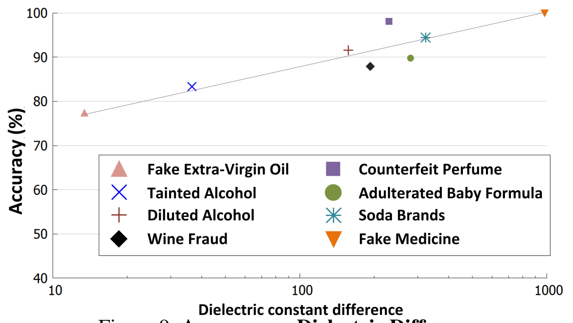


Figure 9: **Accuracy vs Dielectric Differences.**

other environments. In the first regime, the dataset from all environments was mixed and divided into mutually-exclusive training and testing sets. The split-ratio is 80% training and 20% testing for both RF-EATS and the state-of-the-art baseline, RFIQ.[9]

Fig. 7 plots the accuracy for each of the applications in both regimes, and compares them to each of the baselines described in §4.3. We make the following remarks:

- When trained and tested in the same environment, most models achieve high accuracy ($\geq$90%) across all applications. In such scenarios, RF-EATS matches or exceeds the performance of the baselines.
- When testing in new and unseen environments, the accuracy of all the baselines drop significantly, some to a random guess. On the other hand, RF-EATS's accuracy remains around or above 90% for six out of the eight applications, and above 83% for all the applications. Its median improvement over a simple neural network is 15.1%, over RFIQ is 26.5%, and over the ray-tracing model is 29.0% across these applications. This shows that RF-EATS's model can indeed learn representative multipath distributions and generalize to unseen environments.
- The neural network (VAE) trained with a ray-tracing model achieves the worst performance. As discussed in §3.1.1, this is because the ray-tracing model ignores important radio propagation characteristics. As a result, a VAE trained on such a model cannot capture representative distributions and is significantly outperformed by RF-EATS's approach which leverages the multipath kernel.

Next, we would like to gain more insight into RF-EATS's performance as a counterfeit detector. In counterfeit detection, it is important to understand the impact of false postives and false negatives. The standard metrics for quantifying them are denoted by precision and recall [10]:

$$Precision = \frac{TP}{TP+FP}, \quad Recall = \frac{TP}{TP+FN} \qquad (8)$$

where TP is true positive, FP is false positive, and FN is false negative. Fig. 8 shows the precision and recall, demonstrating that they are of the same order of RF-EATS's overall accuracy.[10] Note that these metrics are not reported for soda brand application since false positives/negatives are not meaningful in that context.

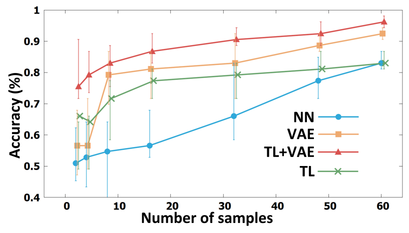
Impact of Dielectric Difference on Accuracy. Next, we would like to confirm that RF-EATS's performance depends on the dielectric differences between the contents of interest. We measured the dielectric of the various contents using the ground truth probe described in §4.3.[11] Because the dielectric constant changes with frequency, we computed the following dielectric distance metric $\sum_{i=1}^{L} \mathcal{L}\left(D_{A,i}, D_{B,i}\right)$ where $\mathcal{L}$ is *L2norm*[12] and

[9]Note that we experimented with different splitting ratios and found that RF-EATS outperformed RFIQ irrespective of the split ratio.

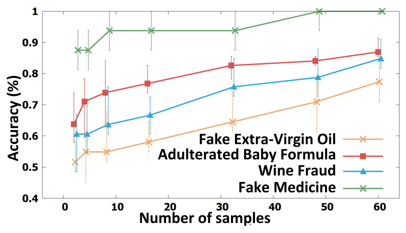
[10]The accuracy plotted in Fig. 7 can be expressed as: (TP+TN)/(TP+TN+FP+FN).

[11]Our measurements are provided in Appendix A.

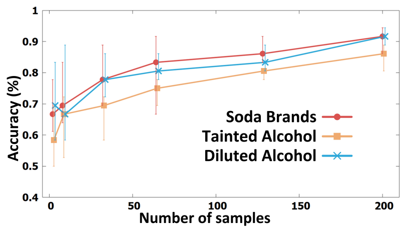
[12]Note that it is possible to use other distance metrics.



(a) Partial Implementations



(b) Performance across Applications



(c) Accuracy for Larger Datasets

Figure 10: **Micro-benchmarks** The figure plots the median accuracy as a function of the training dataset for (a) partial implementations, (b) different applications, and (c) large dataset sizes. Error bars indicate lowest and highest accuracy for different dataset splits between training and testing.

$D_{A,i}, D_{B,i}$ are the dielectric constants at *i-th* frequency component of samples A and B.

Fig. 9 plots the classification accuracy as a function of the dielectric difference for each of the tasks. The figure shows a log-linear relationship between the accuracy and the dielectric differences. When the difference is over 200, RF-EATS achieves 90+% accuracy. The figure also shows that the olive oil application has the lowest accuracy and lowest difference of 15. These results verify that RF-EATS's performance is directly impacted by extent of dielectric differences between the contents of interest.

4.4.2 Micro-benchmarks

We would like to quantify the accuracy gains arising from each of RF-EATS's subcomponents. To do so, we evaluated the accuracy of partial implementations of the overall system: (1) a simple neural network,[13] (2) with the VAE (but no transfer learning), (3) with transfer learning (but no VAE), (4) RF-EATS's full architecture with both the VAE and transfer learning. Similar to our earlier evaluation, we trained on one environment and evaluated on the rest. Additionally, when training the transfer learning model on a given task (e.g., counterfeit perfume), the source domain is obtained by training the encoder on datasets from all the other tasks.

Fig. 10(a) plots the accuracy as a function of the size of the training dataset for each of the above models. For simplicity, the figure only plots results for the counterfeit perfume application . For each dataset size, we randomly chose samples from the database for training, while the rest were used for testing. We repeated this experiment ten times, each time randomly choosing a different subset of samples. The figure plots the median accuracy, and the error bars indicate the maximum and minimum accuracies across the ten iterations.

We make the following remarks:

- Each of the subcomponents contributes to the overall system performance. The improvement in accuracy from the transfer learning classifier over a simple neural network shows that the transfer classifier has higher start point, and VAE has higher slope and asymptote. This is because when the dataset is very small, transfer learning allows extracting well-defined general features and can contribute more to the overall accuracy. However, when the data size becomes large enough, the VAE has higher accuracy because of its effectiveness in generalizing to unseen environments.
- The accuracy gains are more pronounced for smaller datasets. Moreover, the overall system has a smaller standard deviation in comparison to the simple baseline. This result demonstrates the model's ability to not just improve accuracy but also reduce the dataset size required for training.
- The accuracy of each of the models increases with the dataset size. This holds true not only for the perfume classifier but for all classification tasks as shown in Fig. 10(b)-(c), and it is expected because more data enables the learning model to better train its classifier.

[13] We also experimented with networks consisting of more than three layers, but they performed worse. This is likely due to overfitting to the small datasets, so we do not report their results.

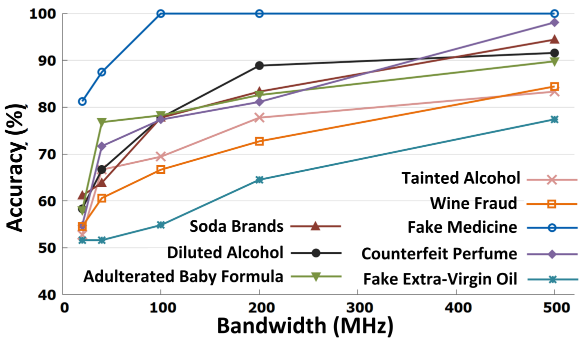


Figure 11: **Impact of Bandwidth on Accuracy.**

Next, we would like to understand the impact of the estimation bandwidth on RF-EATS's accuracy. Recall that our implementation uses two-frequency excitation to sense an RFID's response over a wide bandwidth. To evaluate the impact of bandwidth on accuracy, we used the same measurements from the experimental trials performed earlier, but we only provided varying chunks of bandwidths in training and testing the learning model.

Fig. 11 plots the accuracy on the y-axis as a function of the bandwidth for each of the classification tasks. It shows that RF-EATS's accuracy increases with bandwidth for all of the tasks. In contrast, when using a bandwidth that is constrained to the UHF ISM band (26MHz) as with a standard RFID reader, the accuracies across

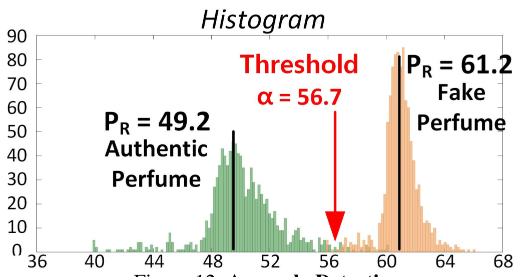


Figure 12: **Anomaly Detection**

most tasks are close to a random guess. This demonstrates the importance of wideband estimation for enabling RF-based food and liquid sensing.

4.4.3 Extending to Unseen Compositions

Finally, we would like to evaluate RF-EATS's ability to perform anomaly detection as per §3.2.2. To do so, we trained the VAE on outputs of the multipath kernel obtained from all datasets except the perfume dataset. Then, we applied the multipath kernel to both the counterfeit and non-counterfeit perfume. Specifically, we compute $h^a_{k,MPATH}/h^a_{k,LoS}$ and $h^a_{k,MPATH}/h^a_{k,LoS}$ where $h^{a/f}_{k,MPATH}$ denotes the channel of authentic/fake perfume in multipath-rich environments, and $h^a_{k,LOS}$ denotes the line-of-sight channel of authentic perfume. The detector is provided with samples of $h^a_{k,LOS}$ knowing that they are authentic. We set $L = 1000$ in Algorithm 3.1, resulting in 1,000 reconstruction loss values.

Fig. 12 shows the histogram of the reconstruction loss of authentic (green) and fake (orange) perfumes. The figure shows that the average reconstruction loss of fake (61.2) is significantly smaller than that of the authentic (49.2) perfume. If we set the threshold value α to 56.7, we would obtain 94% accuracy in counterfeit detection.

5 Other Related Work

(a) RF sensing. RF-EATS is related to a large and emerging body of literature on using wireless signals for sensing purposes, including shape-based object classification [75, 74], radio localization [73, 31], liquid level sensing [18], and human sensing [12, 52, 30]. RF-EATS's goal is orthogonal to these proposals and it focuses on food and liquid sensing. It is worth noting that while some of these proposals can generalize to new multipath environments, their problem statement is fundamentally different than RF-EATS's because the impact of the dielectric and multipath is not linearly separable as explained in §3.1.

(b) Use of Machine Learning in Wireless. Motivated by recent advances in machine learning, wireless researchers have adapted these advances to a variety of communication and sensing tasks, including end-to-end decoding [22], signal classification [48], localization [45, 68, 67, 69, 36], imaging [79], physiological sensing [80], and spectrum monitoring [38, 20]. RF-EATS is similarly motivated by recent advances in learning; however, it focuses on the liquid sensing problem and introduces new contributions that allow it to address domain-specific challenges.

Finally, prior work on human activity sensing has recognized the problem of generalizing RF learning models to different environments [28, 80]. Existing solutions require collecting datasets for each activity and label across a large number of environments. In our context, this would require collecting measurements for each label/contaminant/class across a wide variety of environments, significantly increasing the training and data collection effort over RF-EATS's approach. In contrast, RF-EATS's multipath kernel and transfer learning approach allow it to generalize to new environments and sensing tasks in a sample-efficient manner.

6 Discussion & Conclusion

RF-EATS marks an important step toward ubiquitous, low-cost food sensing using pervasive networking technologies. Our evaluation demonstrated RF-EATS's ability to deliver important applications, its resilience to changing indoor environments, and its efficiency in generalizing to new tasks and unseen environments.

RF-EATS's design and evaluation can be extended in multiple ways. First, while our evaluation focused on changes in the radio environment (multipath reflections, positions, etc.), the RFID measurements may also be impacted by other environmental factors such as the shape and material of the container itself or even the impact of temperature changes on the dielectric. It may be possible to extend RF-EATS's VAE to similarly learn the distributions of such changes and enable it to generalize to these environmental variables as well. We note that even without generalizing the model to different container shapes and materials, it can be used as-is in the context of counterfeit detection (e.g., fake perfume) since such items are typically designed to look very similar to the original products (but have different contents). Another valuable extension of RF-EATS is via miniaturization. Specifically, our current prototype is relatively bulky for direct consumer use. However, the large size is primarily due to the use of USRP software radios and log-periodic antennas for flexibility of prototyping, and can be miniaturized in future design iterations.

As the research evolves, we hope that it can continue bringing low-cost food and liquid sensing closer to the hands of lay consumers to help democratize food and product safety solutions.

Acknowledgments. We thank our shepherd Kyle Jamieson, the anonymous NSDI reviewers, and the Signal Kinetics group for their helpful feedback on the manuscript. This research is supported by a J-WAFS grant, the NSF, and the MIT Media Lab.

References

[1] Alert for methanol. https://foodsafety.neogen.com/en/alert-methanol.

[2] ALN-9640 Squiggle Inlay. www.alientechnology.com. Alien Technology Inc.

[3] Children's tylenol cold & cough. https://www.tylenol.ca/products/infants-children/childrens-tylenol-cold-cough.

[4] Counterfeit soft drinks? https://www.ameribev.org/education-resources/blog/post/counterfeit-soft-drinks/.

[5] E8362B PNA Network Analyzer, 10 MHz to 20 GHz. https://www.keysight.com/en/pd-72279-pn-E8362B/pna-series?cc=US&lc=eng. keySight technologies.

[6] Enfamil neuropro infant formula, ready to use. https://www.enfamil.com/products/enfamil-neuropro-infant/8-fl-oz-ready-to-use-bottles-case-24.

[7] Extra virgin olive oil. https://www.goya.com/en/products/extra-virgin-olive-oil.

[8] Graves grain alcohol 190@ - 750ml. https://www.worldwidebev.com/graves-grain-alcohol-190at-4360.html.

[9] N1501A Dielectric Probe Kit. https://www.keysight.com/en/pd-2492144-pn-N1501A/dielectric-probe-kit?cc=US&lc=eng. keySight technologies.

[10] Precision and recall. https://en.wikipedia.org/wiki/Precision_and_recall.

[11] usrp n210. http://www.ettus.com, 2017. ettus inc.

[12] ADIB, F., HSU, C.-Y., MAO, H., KATABI, D., AND DURAND, F. Capturing the human figure through a wall. *ACM Transactions on Graphics (TOG) 34*, 6 (2015), 219.

[13] AN, J., AND CHO, S. Variational autoencoder based anomaly detection using reconstruction probability.

[14] AN, Z., LIN, Q., AND YANG, L. Cross-frequency communication: Near-field identification of uhf rfids with wifi! In *MobiCom* (2018), pp. 623–638.

[15] ANG, P. K., CHEN, W., WEE, A. T. S., AND LOH, K. P. Solution-gated epitaxial graphene as ph sensor. *Journal of the American Chemical Society 130*, 44 (2008), 14392–14393.

[16] BANERJEE, P., KINTZIOS, S., AND PRABHAKARPANDIAN, B. Biotoxin detection using cell-based sensors. *Toxins 5*, 12 (2013), 2366–2383.

[17] BATTAN, C. How fake beauty products have infiltrated amazon, target, and other reliable retailers, 2017.

[18] BHATTACHARYYA, R., FLOERKEMEIER, C., AND SARMA, S. Low-cost, ubiquitous rfid-tag-antenna-based sensing. *Proceedings of the IEEE 98*, 9 (2010), 1593–1600.

[19] CUMMING, E. The great wine fraud, 2016.

[20] DAVASLIOGLU, K., AND SAGDUYU, Y. E. Generative adversarial learning for spectrum sensing. In *2018 IEEE International Conference on Communications (ICC)* (2018), IEEE, pp. 1–6.

[21] DHEKNE, A., GOWDA, M., ZHAO, Y., HASSANIEH, H., AND CHOUDHURY, R. R. Liquid: A wireless liquid identifier. In *ACM MobiSys* (2018).

[22] FELIX, A., CAMMERER, S., DÖRNER, S., HOYDIS, J., AND TEN BRINK, S. Ofdm-autoencoder for end-to-end learning of communications systems. In *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)* (2018), IEEE, pp. 1–5.

[23] FENG, C., LI, X., CHANG, L., XIONG, J., CHEN, X., FANG, D., LIU, B., CHEN, F., AND ZHANG, T. Material identification with commodity wi-fi devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems* (2018), ACM, pp. 382–383.

[24] GOSSNER, C. M.-E., SCHLUNDT, J., BEN EMBAREK, P., HIRD, S., LO-FO-WONG, D., BELTRAN, J. J. O., TEOH, K. N., AND TRITSCHER, A. The melamine incident: implications for international food and feed safety. *Environmental health perspectives 117*, 12 (2009), 1803–1808.

[25] HA, U., MA, Y., ZHONG, Z., HSU, T.-M., AND ADIB, F. Learning food quality and safety from wireless stickers. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (2018), ACM, pp. 106–112.

[26] HENLEY, J. How to tell if your olive oil is the real thing, 2012.

[27] HUANG, W.-D., CAO, H., DEB, S., CHIAO, M., AND CHIAO, J.-C. A flexible ph sensor based on the iridium oxide sensing film. *Sensors and Actuators A: Physical 169*, 1 (2011), 1–11.

[28] JIANG, W., MIAO, C., MA, F., YAO, S., WANG, Y., YUAN, Y., XUE, H., SONG, C., MA, X., KOUTSONIKOLAS, D., ET AL. Towards environment independent device free human activity recognition. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (2018), pp. 289–304.

[29] JOHNSON, R. C., AND JASIK, H. Antenna engineering handbook. *New York, McGraw-Hill Book Company, 1984, 1356 p. No individual items are abstracted in this volume.* (1984).

[30] JOSHI, K., BHARADIA, D., KOTARU, M., AND KATTI, S. Wideo: Fine-grained device-free motion tracing using {RF} backscatter. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015), pp. 189–204.

[31] JOSHI, K., HONG, S., AND KATTI, S. Pinpoint: Localizing interfering radios. In *Usenix NSDI* (2013).

[32] KARIMI, F. US warns travelers about tainted alcohol in Mexico. CNN, 2017. http://www.cnn.com/2017/07/27/us/mexico-state-department-alcohol-warning/index.html.

[33] KINGMA, D. P., AND WELLING, M. Auto-encoding variational bayes. In *2nd International Conference on Learning Representations (ICLR)* (2013).

[34] KOTARU, M., JOSHI, K., BHARADIA, D., AND KATTI, S. Spotfi: Decimeter level localization using wifi. In *ACM SIGCOMM computer communication review* (2015), vol. 45, ACM, pp. 269–282.

[35] KUSWANDI, B., WICAKSONO, Y., ABDULLAH, A., HENG, L. Y., AHMAD, M., ET AL. Smart packaging: sensors for monitoring of food quality and safety. *Sensing and Instrumentation for Food Quality and Safety 5*, 3-4 (2011), 137–146.

[36] LI, Q., QU, H., LIU, Z., ZHOU, N., SUN, W., AND LI, J. Af-dcgan: Amplitude feature deep convolutional gan for fingerprint construction in indoor localization system. *arXiv preprint arXiv:1804.05347* (2018).

[37] LI, Z., AND SUSLICK, K. S. A hand-held optoelectronic nose for the identification of liquors. *ACS sensors 3*, 1 (2018), 121–127.

[38] LI, Z., XIAO, Z., WANG, B., ZHAO, B. Y., AND ZHENG, H. Scaling deep learning models for spectrum anomaly detection. In *Proceedings of the Twentieth ACM International Symposium on Mobile Ad Hoc Networking and Computing* (2019), ACM, pp. 291–300.

[39] LUO, Z., ZHANG, Q., MA, Y., SINGH, M., AND ADIB, F. 3d backscatter localization for fine-grained robotics. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 765–782.

[40] MA, Y., SELBY, N., AND ADIB, F. Minding the billions: Ultrawideband localization for deployed rfid tags. *ACM MobiCom* (2017).

[41] MAKHZANI, A., SHLENS, J., JAITLY, N., AND GOODFELLOW, I. Adversarial autoencoders. In *International Conference on Learning Representations* (2016).

[42] NARSAIAH, K., JHA, S. N., BHARDWAJ, R., SHARMA, R., AND KUMAR, R. Optical biosensors for food quality and safety assurance: a review. *Journal of food science and technology 49*, 4 (2012), 383–406.

[43] NEW YORK TIMES. China's Top Food Official Resigns. http://www.nytimes.com/2008/09/23/world/asia/23milk.html.

[44] NGUYEN, D. S., LE, N. N., LAM, T. P., FRIBOURG-BLANC, E., DANG, M. C., AND TEDJINI, S. Development of novel wireless sensor for food quality detection. *Advances in Natural Sciences: Nanoscience and Nanotechnology 6*, 4 (2015), 045004.

[45] NIITSOO, A., EDELHÄUβER, T., AND MUTSCHLER, C. Convolutional neural networks for position estimation in tdoa-based locating systems. In *2018 International Conference on Indoor Positioning and Indoor Navigation (IPIN)* (2018), IEEE, pp. 1–8.

[46] ONG, K. G., BITLER, J. S., GRIMES, C. A., PUCKETT, L. G., AND BACHAS, L. G. Remote query resonant-circuit sensors for monitoring of bacteria growth: Application to food quality control. *Sensors 2*, 6 (2002), 219–232.

[47] Ong, K. G., Bitler, J. S., Grimes, C. A., Puckett, L. G., and Bachas, L. G. Remote query resonant-circuit sensors for monitoring of bacteria growth: Application to food quality control. *Sensors 2*, 6 (2002), 219–232.

[48] O'Shea, T. J., Roy, T., and Clancy, T. C. Over-the-air deep learning based radio signal classification. *IEEE Journal of Selected Topics in Signal Processing 12*, 1 (2018), 168–179.

[49] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch.

[50] Polgreen, L. 84 children are killed by medicine in nigeria, 2009.

[51] Potyrailo, R. A., Nagraj, N., Tang, Z., Mondello, F. J., Surman, C., and Morris, W. Battery-free radio frequency identification (rfid) sensors for food quality and safety. *Journal of agricultural and food chemistry 60*, 35 (2012), 8535–8543.

[52] Pu, Q., Jiang, S., Gollakota, S., and Patel, S. Whole-home gesture recognition using wireless signals. In *ACM MobiCom* (2013).

[53] Pu, Y., Gan, Z., Henao, R., Yuan, X., Li, C., Stevens, A., and Carin, L. Variational autoencoder for deep learning of images, labels and captions. In *Advances in neural information processing systems* (2016), pp. 2352–2360.

[54] Purefoy, C. Poisoned medicine kills dozens of children in nigeria, 2008.

[55] Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[56] Rahman, T., Adams, A. T., Schein, P., Jain, A., Erickson, D., and Choudhury, T. Nutrilyzer: A mobile system for characterizing liquid food with photoacoustic effect. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM* (2016), ACM, pp. 123–136.

[57] Rahman, T., Adams, A. T., Schein, P., Jain, A., Erickson, D., and Choudhury, T. Nutrilyzer: A mobile system for characterizing liquid food with photoacoustic effect. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM* (New York, NY, USA, 2016), SenSys '16, ACM, pp. 123–136.

[58] Ruch, P., Hu, R., Capua, L., Temiz, Y., Paredes, S., Lopez, A., Barroso, J., Cox, A., Nakamura, E., and Matsumoto, K. A portable potentiometric electronic tongue leveraging smartphone and cloud platforms. pp. 1–3.

[59] Smartrac Group. Smartrac Shortdipole Inlay. www.smartrac-group.com.

[60] Sridhar, M., and Reddy, C. R. Surface tension of polluted waters and treated wastewater. *Environmental Pollution Series B, Chemical and Physical 7*, 1 (1984), 49–69.

[61] Sun, C., and Trueman, C. W. Unconditionally stable crank-nicolson scheme for solving two-dimensional maxwell's equations. *Electronics Letters 39*, 7 (April 2003), 595–597.

[62] Tan, E. L., Ng, W. N., Shao, R., Pereles, B. D., and Ong, K. G. A wireless, passive sensor for quantifying packaged food quality. *Sensors 7*, 9 (2007), 1747–1756.

[63] Tse, D., and Viswanath, P. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[64] Ulaby, F. T., Michielssen, E., and Ravaioli, U. Fundamentals of applied electromagnetics 6e. *Boston, Massachussetts: Prentice Hall* (2010).

[65] Vasilescu, A., and Marty, J.-L. Electrochemical aptasensors for the assessment of food quality and safety. *TrAC 79* (2016), 60–70.

[66] Wang, J., Xiong, J., Chen, X., Jiang, H., Balan, R., and Fang, D. Tagscan: Simultaneous target imaging and material identification with commodity rfid devices. *ACM MobiCom* (2017).

[67] Wang, X., Gao, L., and Mao, S. Csi phase fingerprinting for indoor localization with a deep learning approach. *IEEE Internet of Things Journal 3*, 6 (2016), 1113–1123.

[68] Wang, X., Gao, L., Mao, S., and Pandey, S. Csi-based fingerprinting for indoor localization: A deep learning approach. *IEEE Transactions on Vehicular Technology 66*, 1 (2016), 763–776.

[69] Wang, X., Wang, X., and Mao, S. Cifi: Deep convolutional neural networks for indoor localization with 5 ghz wi-fi. In *2017 IEEE International Conference on Communications (ICC)* (2017), IEEE, pp. 1–6.

[70] WANG, Y. In Flint, Mich., there's so much lead in children's blood that a stat of emergency is declared. The Washington Post, 2015. https://www.washingtonpost.com/news/morning-mix/wp/2015/12/15/toxic-water-soaring-lead-levels-in-childrens-blood-create-state-of-emergency-in-flint-mich/.

[71] WEI, M., HUANG, S., WANG, J., LI, H., YANG, H., AND WANG, S. The study of liquid surface waves with a smartphone camera and an image recognition algorithm. *European Journal of Physics 36*, 6 (2015), 065026.

[72] WU, S.-Y., YANG, C., HSU, W., AND LIN, L. 3d-printed microelectronics for integrated circuitry and passive wireless sensors. *Microsystems & Nanoengineering 1* (2015), 15013.

[73] XIONG, J., AND JAMIESON, K. ArrayTrack: a fine-grained indoor location system. In *Usenix NSDI* (2013).

[74] YANZI, Z., YIBO, Z., BEN, Y. Z., AND HAITAO, Z. Reusing 60ghz radios for mobile radar imaging. In *ACM MobiCom* (2015).

[75] YEO, H.-S., FLAMICH, G., SCHREMPF, P., HARRIS-BIRTILL, D., AND QUIGLEY, A. Radarcat: Radar categorization for input & interaction. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (2016), ACM, pp. 833–841.

[76] YOON, J.-Y., AND KIM, B. Lab-on-a-chip pathogen sensors for food safety. *Sensors 12*, 8 (2012), 10713–10741.

[77] YUE, S., AND KATABI, D. Liquid testing with your smartphone. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (2019), ACM, pp. 275–286.

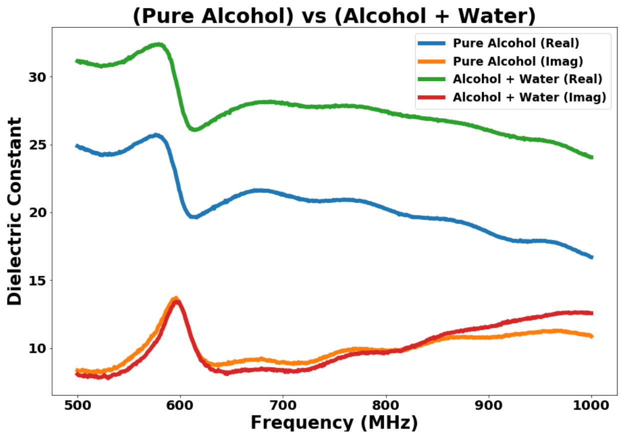
[78] ZHANG, J., AND TIAN, G. Y. Uhf rfid tag antenna-based sensing for corrosion detection & characterization using principal component analysis. *IEEE TAP 64*, 10 (2016), 4405–4414.

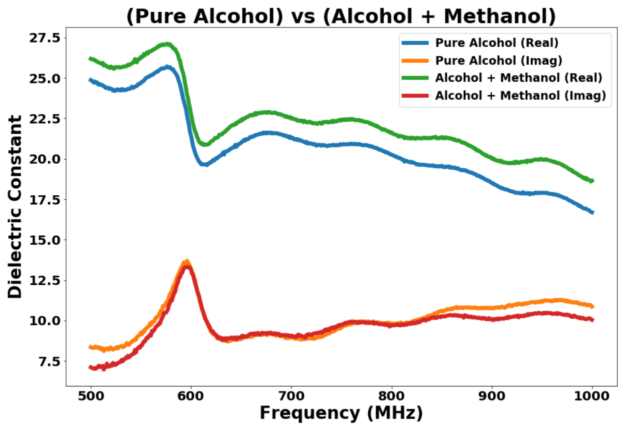
[79] ZHAO, M., TIAN, Y., ZHAO, H., ALSHEIKH, M. A., LI, T., HRISTOV, R., KABELAC, Z., KATABI, D., AND TORRALBA, A. Rf-based 3d skeletons. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), ACM, pp. 267–281.

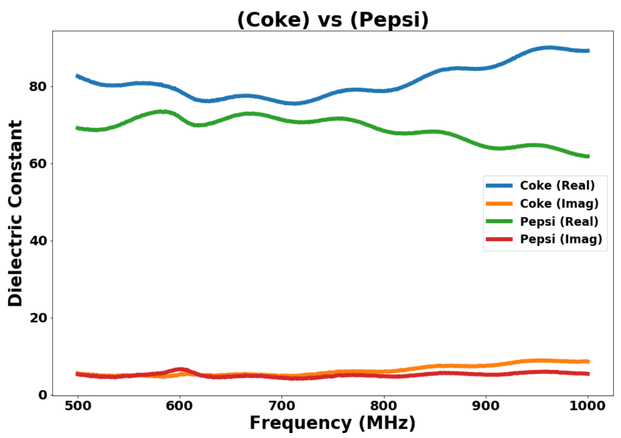
[80] ZHAO, M., YUE, S., KATABI, D., JAAKKOLA, T. S., AND BIANCHI, M. T. Learning sleep stages from radio signals: a conditional adversarial architecture. In *International Conference on Machine Learning* (2017), pp. 4100–4109.

A Dielectric Measurements

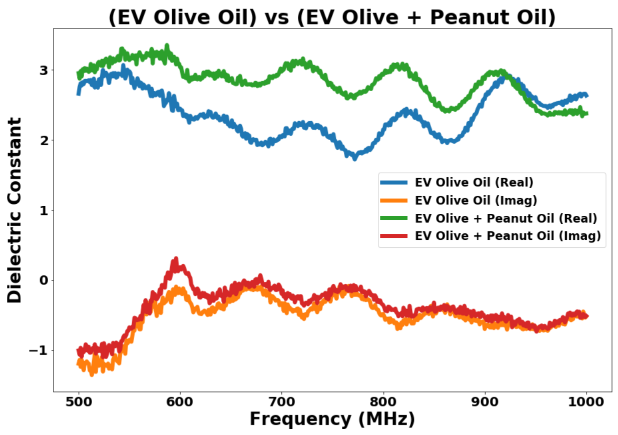
To ensure there is a detectable difference between the real and contaminated/fake products, we measured the dielectric constants of each of the materials. We used the Keysight N1501A Dielectric Probe Kit and E8362B PNA Network Analyzer. We calibrated the dielectric probe with 20°C water and performed a linear sweep from 500MHz to 1GHz, in steps of 1 MHz. The graphs below plot the dielectric constants as a function of frequency.



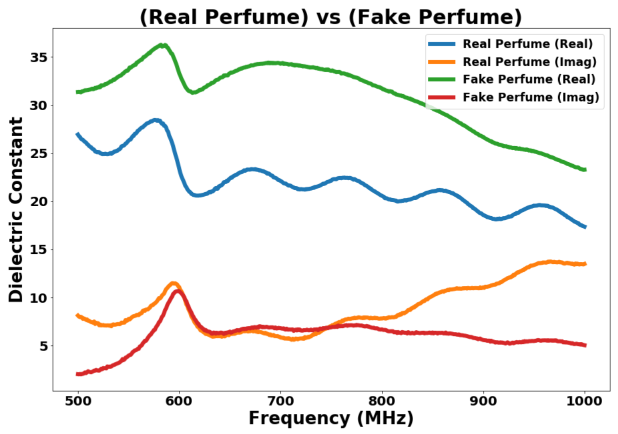




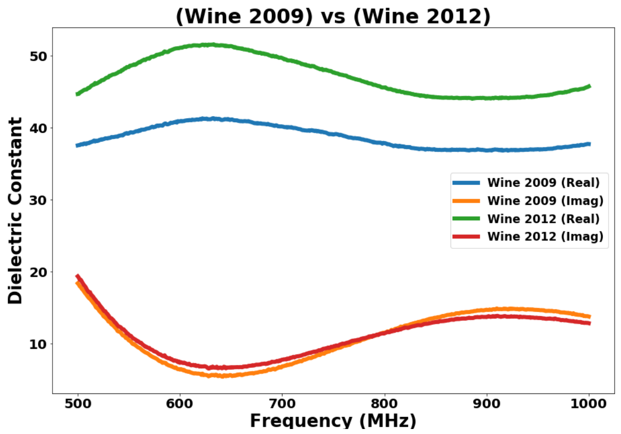
(Coke) vs (Pepsi)
Dielectric Constant
Frequency (MHz)
Coke (Real)
Coke (Imag)
Pepsi (Real)
Pepsi (Imag)



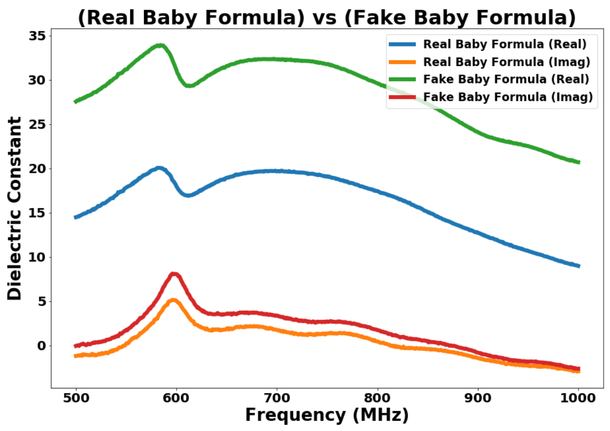
(EV Olive Oil) vs (EV Olive + Peanut Oil)
Dielectric Constant
Frequency (MHz)
EV Olive Oil (Real)
EV Olive Oil (Imag)
EV Olive + Peanut Oil (Real)
EV Olive + Peanut Oil (Imag)



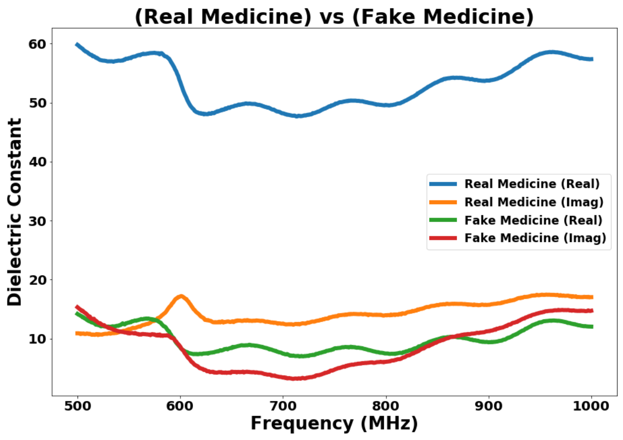
(Real Perfume) vs (Fake Perfume)
Dielectric Constant
Frequency (MHz)
Real Perfume (Real)
Real Perfume (Imag)
Fake Perfume (Real)
Fake Perfume (Imag)



(Wine 2009) vs (Wine 2012)
Dielectric Constant
Frequency (MHz)
Wine 2009 (Real)
Wine 2009 (Imag)
Wine 2012 (Real)
Wine 2012 (Imag)



(Real Baby Formula) vs (Fake Baby Formula)
Dielectric Constant
Frequency (MHz)
Real Baby Formula (Real)
Real Baby Formula (Imag)
Fake Baby Formula (Real)
Fake Baby Formula (Imag)



(Real Medicine) vs (Fake Medicine)
Dielectric Constant
Frequency (MHz)
Real Medicine (Real)
Real Medicine (Imag)
Fake Medicine (Real)
Fake Medicine (Imag)

B Results of Multiclass Classification

Fig. 13 plots the confusion matrix table for multiclass classification between all the tested contents. Training and testing regime performed in different environments similar to Fig. 7(b). The different rows of the matrix represent the actual class of the samples, while the different columns represent RF-EATS's predicted class. The overall accuracy is 85.8%.

A	Alcohol	B	Alcohol+Water
C	Alcohol+Methanol	D	Coke
E	Diet Coke	F	Pepsi
G	Olive Oil	H	Olive Oil + Peanut Oil
I	Baby Formula	J	Adulterated Baby Formula
K	Perfume	L	Fake Perfume
M	Medicine	N	Fake Medicine
O	Wine (2009)	P	Wine (2012)

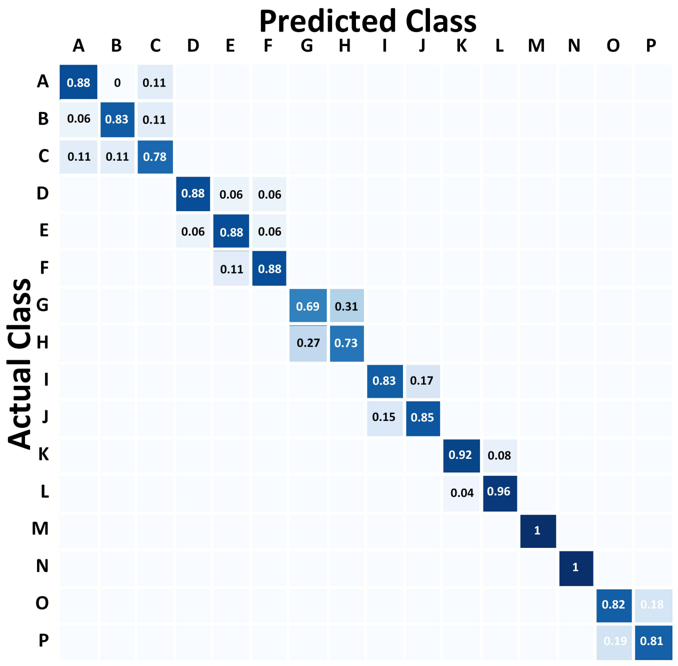


Figure 13: **Confusion Matrix**

Eingerprint: Robust Energy-related Fingerprinting for Passive RFID Tags

Xingyu Chen* Jia Liu* Xia Wang Haisong Liu Dong Jiang Lijun Chen
State Key Laboratory for Novel Software Technology, Nanjing University, China

Abstract

RFID tag authentication is challenging because most advanced cryptographic algorithms cannot be afforded by passive tags. Recent physical-layer identification utilizes unique features of RF signals as the fingerprint to authenticate a tag. This approach is effective but difficult for practical use because it either requires a purpose-built device to extract the signal features or is sensitive to environmental conditions. In this paper, we present a new energy-related fingerprint called *Eingerprint* to authenticate passive tags with commodity RFID devices. The competitive advantage of Eingerprint is that it is fully compatible with the RFID standard EPC-global Gen2, which makes it more applicable and scalable in practice. Besides, it takes the electrical energy stored in a tag's resistor-capacitor (RC) circuit as the fingerprint, which is robust to environmental changes such as tag position, communication distance, transmit power, and multi-path effects. We propose a new metric called persistence time to indirectly estimate the energy level in the RC circuit. A select-query based scheme is designed to extract the persistence time by flipping and observing a flag in the tag's volatile memory. We implement a prototype of Eingerprint with commodity RFID devices without any modifications to the hardware or the firmware. Experiment results show that Eingerprint is able to achieve a high authentication accuracy of 99.4% when three persistence times are used, regardless of device diversity and environmental conditions.

1 Introduction

Radio frequency identification (RFID) is gaining increasing popularity in a wide range of applications, including warehouse inventory [15–17, 34], object tracking [13, 24, 25, 27, 30], and supply chain [22], due to its compelling features, dropping costs, and standardizations. Each RFID tag has a unique digital identity to label tagged items, brings item intelligence to our daily life, and allows the reach of the Internet to include objects as diverse as retail products, library books, debit cards, passports, driver licenses, car plates, and medical devices. In general, the RFID tags fall into two categories: active and passive. Active tags have their own power source and remain active all the time. Compared with the passive tags, they have more computational capabilities and longer read ranges. However, the built-in power source makes them bulky and expensive, which restricts these tags to high-end applications. In contrast, passive tags do not have a built-in power source and are powered by either induction or electromagnetic RF signals of the reader. They have limited computational capabilities and a lower read range than active tags. In spite of these limitations, they are common due to their low cost, small size, and longer life.

In recent years, with the proliferation of RFID systems, the problem of RFID security has attracted increasing attention. A great number of authentication protocols have been proposed to identify the authenticity of a tag [5, 8, 9, 19]. In the nascent stage, the authentication protocols check only the data (e.g., TID [4]) stored in a tag's memory, which is vulnerable to counterfeiting attacks: Adversaries can easily retrieve the data from a genuine tag with a commercial reader and forge a replica by filling its memory with the same data as the genuine tag. To address this problem, some cryptographic approaches are studied. By transmitting the ciphertext rather than the plaintext, the communication channel between a reader and a tag is protected against eavesdropping. However, this approach requires extra hardware components to support high computation overhead, which greatly increases the cost of a passive tag as well as reduces the communication range between the reader and the tag. Hence, it is rarely used by most commercial passive tags.

Motivated by the above limitations, recent research has shifted to physical-layer identification (PLI), which is commonly referred to as RF fingerprinting [10, 11, 20, 33, 35, 36]. It is the process of identifying a device based on transmission imperfections exhibited by its radio transceiver. The key appeal of applying RF fingerprint for authentication is twofold. First, RF fingerprints are unique and unpredictable, such that

*These authors contributed equally to this work they can provide high security guarantees against various protocol-layer attacks. Second, no upgrades of hardware or firmware on existing systems are required, which makes it scalable to the wide use of RFID systems. In spite of this advancement, however, PLI suffers from two problems. First, most PLI-based solutions need a specialized device to detect physical-layer signals, which cannot be deployed in commodity RFIDs. Second, some work is not resistant to environmental or signal acquisition factors, e.g., RF phase values, a widely used metric for RF fingerprinting, heavily relies on the RF channels. Two different measurements of the same tag are very likely to give rise to different RF phases.

In this paper, we explore a brand-new fingerprint called energy-related fingerprint (*Eingerprint*) to authenticate passive tags with commodity RFID systems. Eingerprint takes the electrical energy stored in the tag's circuit as the fingerprint, which is robust to environmental conditions, including tag position, tag orientation, communication distance, transmit power, and multi-path effects. The basic idea is that passive tags do not have any built-in power source and are energized by the electromagnetic RF signals issued by the reader. To ensure proper functioning, a tag needs to store some electrical energy into its microchip, which is equivalent to a resistor-capacitor (RC) charging circuit [38]. Due to manufacturing imperfection, no two tags could ever have exactly the same RC circuit. If we can detect this difference, then we are able to fingerprint each tag as desired.

However, this is not easy. Building the electronic test circuit to physically measure the RC circuit of each tag is infeasible because it destroys the tag's structure and functions. Instead, we use a new metric called *persistence time* to indirectly reflect the RC circuit. The persistence time is the time span from the initial supply voltage when the RC circuit is fully charged decaying to a very low level that cannot afford the tag to run properly, which heavily relies on the RC circuit itself. In other words, if two RC circuits differ from each other, their persistence time is very likely to be different. On the basis of this idea, we design a Gen2-compatible approach to measure the persistence time based on a one-bit inventoried flag of a tag (a one-bit register in a tag's volatile memory). The volatile memory requires power to maintain the stored information. Once the power is cut off (or is lower than a threshold), the stored data are quickly lost. By flipping the inventoried flag and continuously observing its status with Gen-2 compatible commands, we are able to extract the persistence time of a tag. Afterwards, a t-test based model is designed to validate the genuineness of the tag. In addition, instead of individual fingerprinting, we propose a quick and reliable scheme to deal with multiple tags in parallel, which greatly improves the time efficiency of tag authentication. The main contributions of this paper are threefold.

- We explore a new energy-related fingerprint called *Eingerprint* to authenticate passive tags with commodity RFID devices. The competitive advantage of Eingerprint is that it is fully compatible with the RFID standard, which makes it more applicable and scalable for practical use. Besides, it takes the electrical energy as the fingerprint, which is robust to various environmental conditions.
- We propose a new metric *persistence time* to indirectly indicate the energy level stored in a tag's RC circuit. A select-query based scheme is designed to measure the persistence time by flipping and observing a flag in the tag's volatile memory.
- We implement a prototype of Eingerprint in a commercial off-the-shelf RFID system with over 1000 tags. Extensive experiments show that our fingerprinting system is able to achieve a high accuracy of 97.3% and 99.4% when one persistence time and three persistence times are used respectively, without any changes to the hardware.

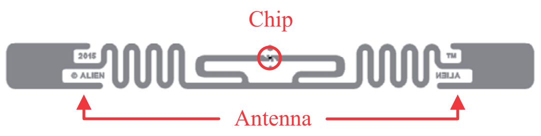


Figure 1: Alien Squiggle general-purpose RFID tag.

The rest of the paper is organized as follows. Section 2 overviews the fingerprinting model and proposes an energy-related fingerprint. Section 3 proposes a Gen2-compatible scheme to derive the fingerprint. Section 4 uses the fingerprint distribution to validate the genuineness of a tag. Section 5 evaluates the performance of the fingerprinting system. Section 6 introduces the related work. Finally, Section 7 concludes this work.

2 Overview

2.1 Fingerprinting Model

Passive tags do not have any built-in power source and are energized by the electromagnetic RF signals emitted by the reader. In general, a passive RFID tag consists of two components: the tag antenna and the microchip. As shown in Fig. 1, the microchip is usually placed right at the terminals of the tag antenna. When the RF signals are received by the tag antenna, the voltage developed on antenna terminals powers up the chip for computing and modulating the backscattered signal. The passive tag can be equivalent to a resistor-capacitor (RC) series circuit [38] that is composed of a resistor and a capacitor.

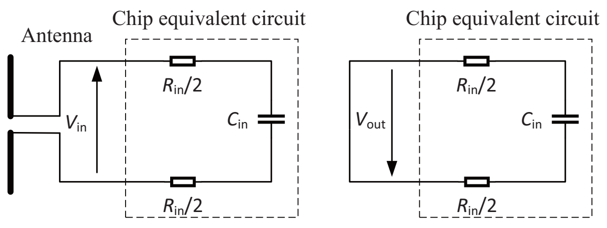
As a result of manufacturing imperfection, no two tags could ever have exactly the same microchip; the same idea applies to the electronic components (the resistor and the capacitor). If we can detect the difference of these electronic components among different tags, then we are able to fingerprint each tag from the physical-layer perspective, which forms the fingerprinting metric of this work. To achieve this goal, an intuitive solution is to set up an electronic test circuit and measure each electronic component manually. This concept works in theory but suffers from three problems in practice. First, a tag needs to be dissected (physically separating the microchip from the antenna), which damages the tag's structure and function. Second, performing measurements individually and manually is time consuming, especially when many tags need to be authenticated. Third, a purpose-built electronic test platform is needed to measure the chip circuit, which increases the cost of fingerprinting and is not scalable in commercial use. Hence, a new fingerprint that is able to reflect the attributes of the electronic components is required.

2.2 Fingerprint: Persistence Time

Consider the RC circuit of the microchip. When the tag captures the energy from the RF signals issued by the reader, it is actually an RC charging process. The equivalent charging circuit is shown in Fig. 2(a), where a capacitor C_{in} in series with a resistor R_{in} is connected across a DC battery supply (the power is obtained from the RF signals). The capacitor will gradually charge up through the resistor until the voltage across it reaches the supply voltage of the DC battery, namely, fully charged. According to the Gen2 standard, this charging process lasts for 2 ms at most. Afterwards, if we remove the voltage source (e.g., turn off the reader) from the fully charged circuit, the capacitor that is able to store the electrical energy acts like a small battery and releases the energy as required. This is referred to as an RC discharging process. As shown in Fig. 2(b), the capacitor discharges through the resistance in the opposite direction, which enables the tag to compute and communicate with the reader. As the discharge continues, the voltage goes down and there is less discharge current across the circuit. When the voltage decays to a very low level that cannot afford the tag to run properly, we say that the tag is exhausted and out of function. Assume that the initial supply voltage of the fully charged circuit is V_{in} and the minimal voltage that is needed to drive a tag is V_0. In the discharging stage, we refer to the time span from the initial supply voltage V_{in} decaying to the voltage threshold V_0 as *persistence time*, which can be derived as follows:

$$T_p = R_{in} \times C_{in} \times \ln(\frac{V_{in}}{V_{in} - V_0}), \quad (1)$$

where R_{in} and C_{in} are the resistance and capacitance of the microchip, respectively [38]. In Eq. (1), the V_0 voltage threshold is a constant when a tag chip is manufactured. For V_{in}, it varies with the available input power and thus depends on the energy captured by the tag antenna. This would be a variable in different communication conditions, e.g., different communication distances. To provide a stable voltage to the digital core, however, the commercial tag is required to carry a low dropout regulator [38], which uses a voltage reference block to produce a regulated and constant voltage V_{in}. Hence, the persistence time relies on the four constants R_{in}, C_{in}, V_0,



(a) Charging. (b) Discharging.

Figure 2: RC circuit of a tag's microchip.

and V_{in}, which are determined by the hardware of a tag, regardless of the environment factors, e.g., the communication distance, the tag location, multipath effects. By measuring the persistence time, we are able to figure out the difference of tag chips. This forms the basic idea of our method.

The energy-based fingerprint has three competitive advantages. First, any Gen2-compatible readers are able to measure the persistence time of a commodity tag with no need for any modifications to the hardware or the firmware. Hence, implementing and deploying the fingerprinting system is easy in practice. Second, the persistence time not only accurately reflects the RC circuit of the tag chip but is also robust to the environment changes (e.g., the communication distance, the tag location, multipath effects), which is a key challenge for some PLI work [11,28]). Third, a commercial tag has several independent persistence times (different RC circuits), which form different fingerprints to jointly authenticate the tag, thus making it hard to counterfeit. In spite of this advancement, measuring the persistence time of a tag is not easy. Next, we first show the system architecture of our approach and then detail how to obtain the persistence time in a Gen2-compatible commodity RFID system.

2.3 System Architecture

In general, the workflow of the fingerprinting system consists of three steps, which are shown in Fig. 3.

- *EPC Identification*: The reader interrogates a tag according to the Gen2 protocol and checks whether the EPC (i.e., tag ID) is identical to the tag to be authenticated. If no, then the tag is counterfeit. Otherwise, the system moves to the second step.
- *Fingerprint Extraction*: This step aims to extract the persistence time of the tag and treats it as the tag's energy fingerprint. Two key issues need to be solved. First, how can the persistence time be measured with the commercial RFID devices? Second, how can the time efficiency be improved and how can the measurement of multiple tags be conducted in parallel?
- *Genuineness Validation*: The system measures the persistence time and validates it with the stored records in the database. If it passes the authentication, then the tag is considered a genuine tag; otherwise, it is a counterfeit.

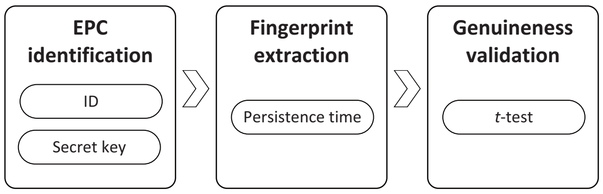


Figure 3: The workflow of the fingerprinting system.

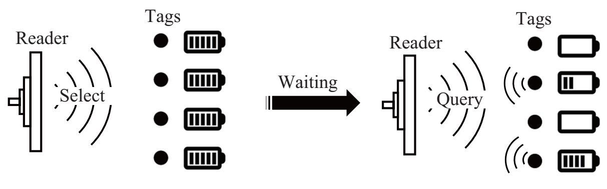


Figure 4: Basic idea of fingerprint extraction.

3 Fingerprint Extraction

3.1 Basic Idea

The basic idea of measuring the persistence time is to build a fully charged RC circuit and then run the discharging operation. This approach requires the following three steps. As shown in Fig. 4, first, we turn on the reader and let it issue the RF signals to energize the tag. Second, after the tag is fully charged, we cut off the power by turning off the reader; the discharging process starts. Third, after a period of time T_d, we check whether the tag is exhausted or not. By gradually increasing the time period T_d and repeating the above three steps, we can find a maximum of T_d that is guaranteed to help the tag work properly. This maximum is actually the persistence time to be measured.

Among above three steps, the first two, turning on and off the reader, are easy to operate. However, examining when the power of the tag is exhausted with a commodity RFID system is challenging. To address this problem, we resort to the volatile memory of a tag. Unlike non-volatile memory (e.g., NAND flash and solid-state drives), the volatile memory requires power to maintain the stored information. Once the power is cut off (or lower than a threshold), the stored data are quickly lost. In the RFID standard Gen2, we find a metric *inventoried flag*, which is a one-bit indicator in a tag's volatile memory. By flipping the inventoried flag and checking its status continuously, we are able to know when the power of the tag is exhausted. Next, we first introduce the Gen2 protocol and then detail how to measure the persistence time based on the RFID standard.

3.2 EPCglobal Gen2 Protocol

The EPCglobal Gen2 (Gen2) protocol is a worldwide UHF RFID standard that defines the physical interactions and logical operating procedures between the readers and tags [4]. On the basis of Gen2, we highlight the related functions that we will be involved by Eingerprint below.

Tag Memory. Gen2 standard specifies that the tag memory is supposed to contain four distinct memory banks (page 44—51 in [4]). MemBank-0 is reserved for kill and access passwords if encryption is implemented on the tag. MemBank-1 stores the electronic product code (EPC), i.e., tag ID that is often referred to. MemBank-2 stores TID that indicates the tag- and manufacturer-specific data at the time of manufacture, which is permalocked and unchangeable. MemBank-3 is user memory that allows customized data storage. In this work, we need to visit the tag's ID, so MemBank-1 is used.

Sessions & Inventoried Flags. Gen2 requires the readers and tags to provide four sessions (denoted as S0, S1, S2, and S3). Tags in one of these sessions shall neither use nor modify an inventoried flag for a different session. This allows two or more readers to use different sessions to independently inventory a common tag population (in different time slots). The inventoried flag is actually a one-bit indicator of a tag's volatile memory. The binary state of the inventoried flag is denoted by A and B, respectively, where A is the initial state as usual. The volatile memory requires power to maintain the stored information. Once the power is lower than a threshold, the stored data are quickly lost, that is, the inventoried flag will flip to A when the power of the tag is exhausted, no matter what the previous state is. According to Gen2, each session corresponds to an independent inventoried flag that needs different power levels to maintain its state, so the persistence time of each inventoried flag is different. Table 1 shows the persistence periods of different sessions specified by the Gen2 protocol. As we can see, when the tag is not energized, the persistence times of the inventoried flags in S2 and S3 are greater than 2 s. In contrast, the persistence time in S1 varies between 500 ms and 5 s, and no persistence time (always in A) is found for S0. This specification provides us with three persistence times by using the inventoried flags in S1, S2, and S3. Next, we take the inventoried flag in S1 as an example to show how our fingerprinting system works; the two other sessions can be used in the same way.

Select. *Select* is a mandatory command that is prior to each inventory round. It allows a reader to choose a specific subset of tags that participate in the subsequent inventoried round.

Table 1: Persistence time

G2 Session	Persistence time
S0	Tag energized: Indefinite Tag not energized: none
S1	Tag energized: 500 ms -5 sec Tag not energized: 500 ms -5 sec
S2	Tag energized: Indefinite Tag not energized: >2 sec
S3	Tag energized: Indefinite Tag not energized: >2 sec

Aside from tag selection, the *Select* command can also assert or deassert a tag's selected (SL) flag, or set a tag's inventoried flag to either A or B. These flags are used to determine whether or not a tag may respond to a reader. Specifically, a *Select* command consists six fields.

• *MemBank, Mask, Length, Pointer*. These four fields jointly determine which tags are matching or not. *MemBank* specifies which memory bank is chosen for comparison. As aforementioned, four memory banks are available, MemBank-0, MemBank-1, MemBank-2, and MemBank-3, which are indicated by 0, 1, 2, and 3, respectively. *Pointer* indicates the starting position in the chosen memory bank. *Length* determines the length of *Mask*, which is a customized bit string according to the user demands. If *Mask* is the same as the string that begins at *Pointer* and ends *Length* bits later in the memory of *MemBank*, then the corresponding tag is matched.

• *Target, Action.* The field *Target* indicates the object that *Select* will operate, which is either a tag's SL flag or an inventoried flag in any one of four sessions. The sessions are specified by the Gen2 protocol to fit the case of exclusive reading among multiple readers. Therefore, five different targets can be chosen. The selection function is actually achieved by masking the interested tags, setting the matching tags' inventoried flags or SL flag to a specific state while not-matching tags to opposite, and finally operating the tags with the same flag state. How to set the inventoried flag and the SL fag is determined by the *Action* field. As shown in Table 2, eight actions are available, where matching and not-matching tags set their inventoried flags to *A* or *B*. By combining *Target* and *Action*, the reader is able to modify the state of the inventoried flags or the SL flag for a group of tags. For example, when the *Action* is 0, the matching tags are set to *A* while the not-matching tags are set to B. The term "do nothing" means the tags keep their flags unchanged.

Query. *Query* command starts a new inventory round over the tag subpopulation, which are chosen by the previous *Select* command(s). In the inventory round, the reader will carry out a frame that consists of some time slots. Each "selected" tag randomly picks one of these time slots and transmits its tag ID to the reader in that slot. After a tag is queried by the reader, it will invert its inventoried flag, i.e., from the state *A* to *B*, or vice versa. *Query* includes three fields that we would like to focus on.

• *Session, Target*. Similar to that in *Select*, this field *Session* in *Query* specifies one of the four sessions used in the incoming inventory round. The field *Target* determines which tags will participate in the current inventory round, where 0 indicates the tags with the inventoried flag being *A* and 1 indicates *B*.

Table 2: Eight actions of *Select*.

Action	Tag Matching	Tag Not-Matching	Abbr.
000	assert **SL** or **inventoried** → *A*	deassert **SL** or **inventoried** → *B*	AB
001	assert **SL** or **inventoried** → *A*	do nothing	A-
010	do nothing	deassert **SL** or **inventoried** → *B*	-B
011	negate **SL** or *(A→B, B→A)*	do nothing	S-
100	deassert **SL** or **inventoried** → *B*	assert **SL** or **inventoried** → *A*	BA
101	deassert **SL** or **inventoried** → *B*	do nothing	B-
110	do nothing	assert **SL** or **inventoried** → *A*	-A
111	do nothing	negate **SL** or *(A→B, B→A)*	-S

• *Sel*. This field consists of two bits that determine which tags respond to *Query*: 00_2 and 01_2 indicate all matching tags in the previous Select command; 10_2 indicates tags with deasserted SL flag ($\sim SL$); and 11_2 indicates tags with asserted SL flag (SL).

On the basis of the above Gen2-compatible functions, we next detail how to jointly utilize the *Select* and *Query* commands to measure the persistence time by using the state of the inventoried flag. The method is called select-query based measurement.

3.3 Select-Query based Measurement (SQM)

The basic idea is that when the internal energy of a tag is exhausted, the inventoried flag will move back to the initial state *A* for sure, regardless of its previous state. If we set the tag's inventoried flag to *B* and keep the RC circuit fully charged, then the time period from starting discharging to the time when the inventoried flag turns to *A* can be treated as the persistence time.

3.3.1 Design of SQM

To measure the discharging time, we need to jointly use the *Select* command and the *Query* command. According to Gen2, a *Select* command can be written as follows:

$$S(\underbrace{t}_{\text{Target}}, \overbrace{a}^{\text{Action}}, \underbrace{b}_{\text{Membank}}, \overbrace{p}^{\text{Pointer}}, \underbrace{l}_{\text{Length}}, \overbrace{k}^{\text{Mask}}). \tag{2}$$

To set a tag's inventoried flag to *B*, the reader just needs to broadcast a *Select* as follows:

$$\text{Flag} \leftarrow \text{BA: } S(1,4,1,32,96,id), \tag{3}$$

where $t = 1$ (001_2) means the operating object is set to the inventoried flag in session 1 (S1), $a = 4$ indicates that the inventoried flags of matching tags will be set to *B*, while those of not-matching tags will be set to *A*, $(b,p,l,k) = (1,32,96,id)$ means the tag's ID is the same as *id* is selected (matching). Note that the first bit of the tag ID starts from the 32nd bit ($p = 32$) in MemBank-1, because the first 32 bits are a protocol-control (PC) word and the tag ID follows behind the PC word. More details can be seen in [4].

By this means, the target tag is set to *B*. Now the question is how long we can obtain a fully charged RC circuit. Gen2 specifies that the charging time should be no longer than 2 ms, which is much less than the time period (about 20 ms) for broadcasting a select command. In other words, once the select command in 3 is carried out, the target tag has the inventoried flag being B and also the RC circuit being fully charged.

Afterwards, we move to the discharging process by turning off the readers. Given that the tag cannot harvest energy from the reader anymore, the stored electric energy is consumed gradually. After a period of time T_d for discharging, the reader broadcasts a query command to check whether any tag with the inventoried flag B exists. The query command is as follows:

$$\text{Query } B: \ Q(Session = 1, Target = 1, Sel = 0). \quad (4)$$

If a tag reply is received, it means that the persistence time of this tag is longer than T_d. In this case, we need to increase T_d by a small step Δ_t and repeat the above select-query process again. For the first time period T_d that makes no tag reply, it is treated as the persistence time to be measured. That is because no tag reply means that the tag's inventoried flag has flipped to A since the power ran out. Note that, for the session 1 (S1), since the persistence time is bounded between 500 ms to 5 s, we can initialize T_d with 500 ms and increases it gradually until no tag reply occurs.

3.3.2 Multiple Tags

So far, we have discussed how to obtain the persistence time of a session for a single tag. In a practical scenario, however, authenticating multiple tags at a time is common. One intuitive solution is to fingerprint each tag in sequence, one by one. This works but suffers from high time latency. To make SQM more efficient and scalable to the multi-tag case, we need to deal with multiple tags in parallel.

An important observation is that broadcasting the select and executing the query operation take only a few milliseconds; the majority of the time overhead comes from trying the waiting period T_d. If we can let multiple tags wait concurrently, the execution time will decline sharply. Following this idea, we first set all target tags' inventoried flags to the state B, instead of individually dealing with one tag at a time. Afterwards, these target tags move to the discharging process and the energy is consumed gradually. After a period of time T_d, we query the tags with flag B as is. If a tag does not respond to the reader, its persistence time is T_d. This process repeats until all target tags are measured. In this way, the long discharging process executes in parallel, which saves a large amount of time overhead. For example, assume that we fingerprint 10 tags in parallel. We can reduce the waiting periods by about 90%; the global authentication performance is much better than the individual authentication performance.

Now the question is how to select a subset of tags and set their inventoried flags to the state B. Assume there are n tags, in which m tags are target tags. We can separate these m tags from the entire tag set via m select commands. The selection process is executed as follows. We first use the $Action = BA$ to select the first tag t_1, i.e., t_1's inventoried flag is set to B while others are A. Afterwards, for the ith tag t_i, the $Action$ is set to $B-$. We use $B-$ because this action will set the matching tag t_i to B accordingly but not change the settings of the previous tags. The commands are shown below.

$$\begin{aligned} &① \ t_1 \leftarrow BA: S(2, a = 4, 1, 32, 96, id_1) \\ &ⓘ \ t_i \leftarrow B-: S(2, a = 5, 1, 32, 96, id_i), \ \ i \in [2, m], \end{aligned} \quad (5)$$

where id_i represents the tag t_i's tag ID, $a = 5$ means the action $B-$, which can be seen in Table 2. Besides, by investigating commodity RFID readers through their data sheets and real experiments, we find that these readers allow multiple selects to be broadcast in one transmission, e.g., two by Impinj R420 [2] and four by ALR 9900+ and ALR F800 [1]. With this function, we are able to fill several selects into a single one, further saving the communication overhead.

3.4 Enhanced SQM

Although concurrently fingerprinting multiple tags can sharply shorten the authentication time, a large gap still exists between SQM and efficient authentication primarily because that the process of increasingly adjusting the waiting time T_d is time-consuming. For example, assume a tag's persistence time is 3 s and the step length is 0.1 s. The waiting time T_d is initialized to 0.5 s and SQM needs to iteratively try 0.5 s, 0.6 s, 0.7 s, ..., 3.0 s. Summing up the overhead of each try, we have the overall time cost 45.5 s. This time cost is fine for some applications without real-time requirements. However, in the applications such as access control systems, this time is too long to be applicable for practical use.

The basic reason for the low time efficiency is that when a waiting time T_d is examined, we need to reset all tags and retry the next one. A longer time is needed for checking. If we can run the measurement within only one waiting time window, the performance will be improved greatly. Through extensive experiments, we find that the query command does not charge the tag in the session S1. In other words, during the discharging process, we are able to keep querying the tags, with no need to turn off the reader. Once a tag is queried by the reader, it will be recharged again.

3.4.1 Design of Enhanced SQM

With these features, the enhanced SQM measures the tag t_1's persistence time as follows. First, similar to the basic SQM, the reader broadcasts a select command (see Eq. (3)) with action BA to set t_1's inventoried flag to B. After that, the discharging process starts and the reader queries the tag with the flag state being A. The query command is

$$\text{Query } A: \ Q(Session = 1, Target = 0). \quad (6)$$

As shown in Fig. 5, during the discharging process, the internal circuit energizes the tag and keeps the inventoried flag

B, so the reader cannot receive any response from the tag t_1. When the power level is too low to maintain the information of the volatile memory, the inventoried flag moves back to the initial state A. At that time, because the reader keeps querying tags with A, the tag t_1 satisfying this condition will reply to the reader. By observing the time span from the start of discharging to the tag reply, we are able to derive the persistence time of the tag. Clearly, enhanced SQM does not need to try different waiting times; only one time window is able to measure the persistence time, which saves a great number of overheads. For example, consider the above tag with 3 s persistence time. Enhanced SQM results in great performance improvement, reducing the time from 44.5 s to only 3 s, in comparison to the basic SQM.

After responding to the reader, the tag flips its inventoried flag to B (according to Gen2); meanwhile, the RC circuit is fully charged. With the reader continuing to query A, the tag will reply after another persistence time. Hence, if we need multiple measures of persistence time, we just need to record each time interval between two adjacent tag responses, which is shown in Fig. 5. In fact, we can also simplify the enhanced SQM by removing the select command, that is, the reader directly enters the inventory stage. By keeping querying tags with A, the reader is able to get each tag's replies. The time interval between any two adjacent tag replies is the tag's persistence time. In addition, enhanced SQM can be extended to the multi-tag case, with no need for any modifications to the measurement process.

3.4.2 Multiple Tags

In spite of advancements, the enhanced SQM faces a new challenge in which the tags beyond the target tags might have negative effects on the measurement of the persistence time, especially when a great number of these tags exist. More specifically, assume that the tag set is τ and $\tau' \subseteq \tau$ is a subset of tags to be authenticated. The problem is that, when we set the inventoried flags of τ' to B with the select command, the tags in $\tau - \tau'$ will be set to A. In the follow-up inventory stage, the reader queries tags with flags being A; these tags $\tau - \tau'$ will attend to respond. As a result, the tags in τ' cannot give a prompt reply when their flags move back to A due to lack of energy. Setting $\tau - \tau'$ to B initially does not work either because these tags will still reply to the reader after their power level is lower than a threshold.

To address this problem, we resort to another indicator: SL flag. As mentioned previously, the SL flag has two states denoted by SL and $\sim SL$. The reader can specify a set of tags in one of the two states, which will participate in the inventory round. The SL flag and the inventoried flag are independent and can be jointly used to remove the interference of $\tau - \tau'$. The solution is to set the target tags τ' to SL while others $\tau - \tau'$ to $\sim SL$. In the inventory stage, we let only the tags with SL participate in the response. By this means, even if

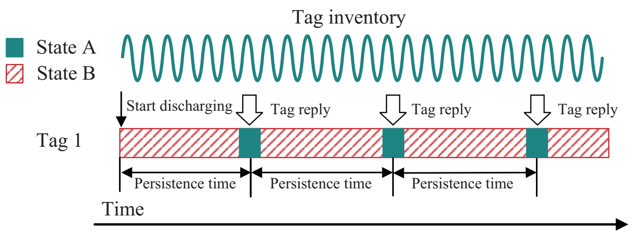


Figure 5: Enhanced SQM for obtaining the persistence time.

a tag in $\tau - \tau'$ is with the inventoried flag A, it has to keep silent to the command of querying A. Specifically, assume that $\tau' = \{t_1, t_2, ..., t_m\}$. The reader broadcasts the select commands as follows:

$$\begin{aligned} &\textcircled{1}\ t_1 \leftarrow AB : S(t = 4, 0, 1, 32, 96, id_1), \\ &\textcircled{i}\ t_i \leftarrow A- : S(t = 4, 1, 1, 32, 96, id_i), \quad i \in [2, m], \end{aligned} \tag{7}$$

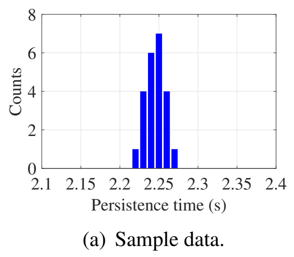
where $Target$ being set to 4 ($t = 4$) means that the operating object of the select is the SL flag. With the above select commands, the SL flags of the tags in τ' are asserted (SL), whereas those of tags in $\tau - \tau'$ are deasserted ($\sim SL$). Afterwards, we move to the inventory stage with the query command:

$$\text{Query } A \ \& \ SL: \ Q(1, Target = 0, Sel = 3), \tag{8}$$

where the fields $Sel = 3$ and $Target = 0$ mean that the reader queries only the tags with the inventoried flags being A together with asserted SL. In such a context, only the target tags of τ' have the chance to reply; other tags in $\tau - \tau'$ are silenced due to $\sim SL$. For any target tag, by recording the time interval between two adjacent replies, we can get its persistence time, which is treated as the energy-related fingerprint.

3.5 Degree of Parallelism

Simultaneous authentication of multiple tags greatly saves the time overhead. However, this is not free; it lowers the sampling rate of each tag when measuring its persistence time. That is because the read throughput of a reader model (how fast the reader can read the tags) is usually fixed. More tags correspond to reduced likelihood that a tag is read. A low sampling rate means a low resolution of measured persistence time, which further affects the authentication accuracy. To address this problem, we can partition a tag set into several small subsets if a large number of tags are to be authenticated. Afterwards, we deal with each subset of tags at a time. The process of fingerprinting a subset of tags can be seen in Section 3.3.2 (for SQM) and Section 3.4.2 (for enhanced SQM). This process repeats until all tags are validated. Note that the degree of parallelism is related to the read throughput of a reader. High read throughput ensures that more tags can be fingerprinted simultaneously. The degree of parallelism is evaluated in Section 5.3.2.



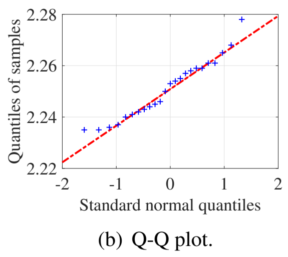


Figure 6: Gaussian distribution of persistence time.

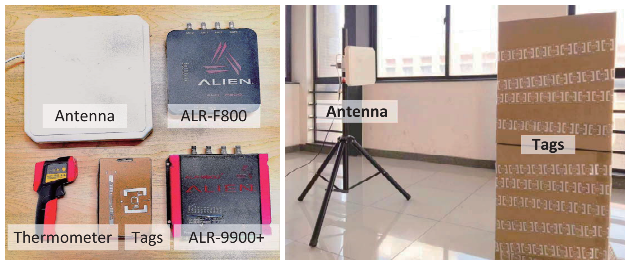


Figure 7: System deployment.

4 Genuineness Validation

To validate the genuineness of a tag, we need to compare its fingerprints under testing with those in the check-in stage. In this work, we take the distribution of the persistence time as the metric to perform the comparison. Intuitively, if a tag under testing is genuine, then its persistence time should follow the same distribution as its genuine records. Given a newly measured set X' of the persistence time and a genuine record X, the task of genuineness validation is reduced to verify whether X' and X follow the same distribution.

Now, we set up an RFID system that contains 1000 tags with eight different models supplied by three leading RFID companies: Alien [1], NXP [3], and Impinj [2]. For each tag, we run the enhanced SQM to obtain at least 20 measures of the persistence time. In Fig. 6(a), we randomly pick a tag and plot its persistence time. As we can see, the persistence time is likely to be a Gaussian distribution. We validate this conclusion through a quantile-quantile (Q-Q) plot, which is widely used to compare the similarity between two probability distributions. If two compared distributions are similar, then the points in the Q-Q plot will nearly lie on a line. As shown in Fig. 6(b), we compare the persistence time with the standard normal distribution. Clearly, the plots almost form a straight line, suggesting that the persistence time follows a Gaussian distribution.

In statistics, t-test is most commonly applied to determine whether the means of two sets of data with Gaussian distribution are significantly different from each other. Suppose the recorded data X follows a Gaussian distribution $N(\mu, \delta^2)$ and the data X' under testing follows a Gaussian distribution $N(\mu', \delta'^2)$. If the tag is a genuine tag, then μ' and δ'^2 are supposed to be very close to μ and δ^2, respectively. According to t-distribution, the mean value $\bar{X}'$ shall be

$$f(\bar{X}') = \frac{\bar{X}' - \mu}{\delta/\sqrt{n}}. \tag{9}$$

The t-test uses the significance level p as a threshold to determine whether or not accept $\bar{X}'$. The significance level p belongs to the interval [0, 1] and is typically set to 0.05 or less [23]. The setting of p will be discussed in Section 5.3.1.

Note that if the persistence time does not follow normal distribution, we can resort to a non-parametric test, e.g., Wilcoxon rank-sum test, which is valid for both non-normally distributed data and normally distributed data.

5 Implementation & Evaluation

In this section, we implement a prototype of Eingerprint in a commodity RFID system. On the basis of this system, we evaluate the performance of Eingerprint through extensive experiments in terms of the robustness to environmental changes and authentication accuracy.

5.1 System Deployment

The system setup is shown in Fig. 7. Two reader models, ALR-F800 and ALR-9900+ supplied by Alien [1], are employed in our experiment without any modifications to the hardware or the firmware. The reader is connected to a directional antenna (Laird S9028 [14], with a gain of 8.5 dBi) and operates at around 920 MHz. Over 1000 tags with 8 tag models are used in total. The model ALN-9634 [1] is adopted as the default in the experiments without explicit instructions. The development software of the fingerprinting system is Java, which adopts the Low-Level Reader Protocol (LLRP), specified by EPCglobal in its EPC Gen2 standard, to communicate. The host computer is a laptop with an Intel Core i5-8250U 1.8 GHz CPU and 8 GB RAM.

5.2 Impact of Environmental Factors

Resilience to environmental conditions is where Eingerprint shines, which is a basis for practical use. In this subsection, we investigate the impact of environmental factors on the measure of the persistence time, including the communication distance, tag orientation, communication frequency, transmit power, and temperature. All results are evaluated based on the inventoried flag in session 1 (S1) without explicit instructions. Similar conclusions can also be drawn in session 2 (S2) and session 3 (S3).

Distance. The communication distance between a reader and a tag is well known to have a great impact on the RF signals, e.g., RSSI or the phase value. To investigate the impact of the distance on Eingerprint, we vary the distance d and

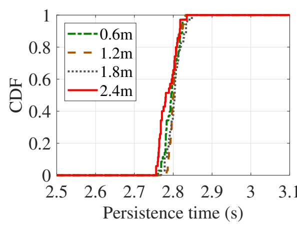


Figure 8: Impact of distance.

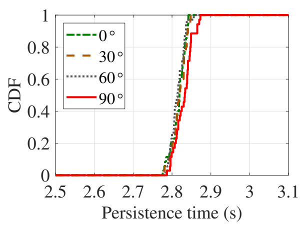


Figure 9: Impact of tag orientation.

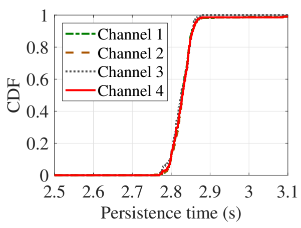


Figure 10: Impact of channel.

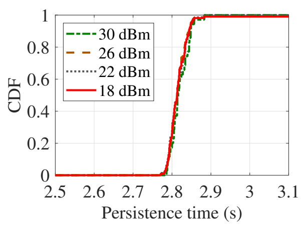


Figure 11: Impact of transmit power.

observe the changes in CDFs of persistence times. In this experiment, four distances are tested, where d_1 =0.6 m, d_2 =1.2 m, d_3 =1.8 m, and d_4 =2.4 m. As shown in Fig. 8, we can see that the CDFs are very close to each other and the means of the persistence times under the four distances are 2.797 s, 2.801 s, 2.804 s, and 2.787 s, respectively. These positive results demonstrate that the distance between a reader and a tag has little effect on the energy-related fingerprint.

Tag orientation. In some existing RF-based work [10,35], the authentication accuracy largely depends on the tag orientation. In Fig. 9, we observe the persistence time of a tag under different rotation angles, i.e., $0°, 30°, 60°, 90°$. The means of the measured persistence times are 2.817 s, 2.818 s, 2.814 s, and 2.814 s, respectively, corresponding to the four rotation angles. Similarly, the consistent results indicate that our energy-related fingerprint remains stable, regardless of the tag's rotation angles.

RF channels. A typical UHF reader has 16 channels working at 920—924 MHz ISM band. RF phase values, a widely used metric for RF fingerprinting, heavily rely on the RF channels. To examine whether the channel affects the stability of Eingerprint, we extract the persistence time from a tag under four different channels, where $channel_1$ = 920.625 MHz, $channel_2$ = 921.625 MHz, $channel_3$ = 922.625 MHz, $channel_4$ = 923.625 MHz. Fig. 10 shows the CDFs of the persistence times under the four channels. The close results demonstrate that the energy-related fingerprint is resistant to the communication channel.

Transmit power. Next, we examine the effect of the transmit power of the reader. In this experiment, we set the transmit power to 30 dBm, 26 dBm, 22 dBm, and 18 dBm respectively, and observe its impact on persistence time. As shown in Fig. 11, the CDFs of the persistence times under different transmit powers approach to each other. The positive results demonstrate that the transmit power has little impact on the energy-related fingerprint.

Temperature. In this experiment, we study the impact of the temperature on the persistence time. Four temperatures are investigated, three of which are close to each other (20 °C, 21 °C, 22 °C) and another is much higher (30 °C). Fig. 12 shows the CDFs of the persistence time under these four temperatures. As we can see, the three CDFs of temperatures 20 °C, 21 °C, 22 °C are similar, while that of 30 °C is different from the others'. This result indicates that the temperature has an impact on the persistence time. If the temperature change is slight, the impact could be negligible. Otherwise, we need to take the temperature into account if it varies considerably. This accords with the theory as temperature could affect the resistance and capacitance of electronic components. In fact, it is a blessing in disguise to some extent: each temperature corresponds to a fingerprint, which provides us with more fingerprints and higher authentication accuracy.

On top of the above experiments, we draw the conclusion that the energy-related fingerprint is resistant to various environmental conditions, including communication distance, tag orientation, communication frequency, and transmit power, except for the temperature.

5.3 Authentication Performance

In the experiments, three widely used metrics are applied to evaluate the authentication performance of Eingerprint, including false acceptance rate (FAR), false rejection rate (FRR), and authentication accuracy. FAR indicates the likelihood that the system will incorrectly accept a counterfeit. FRR indicates the likelihood that the system will fail to accept a genuine tag. For each experiment, we randomly pick two tags from 200 tags and treat one of them as a genuine tag and the other as a counterfeit. By checking whether each of them is genuine or not, we can record the number of correct checks. Repeating the above experiment 500 times, we derive the authentication accuracy that is equal to the ratio of the number of correct checks to the number of tests in total.

5.3.1 Significance Level

Eingerprint utilizes the significance level (threshold), denoted by p, to determine whether a testing fingerprint is valid or not. A large p is likely to reject a valid tag, leading to a high FRR, while a small p cannot figure out friend (genuine tag) or foe (counterfeit), increasing FAR. This dilemma requires a proper value of p to balance FRR and FAR. We extract fingerprints from 200 tags and respectively compute FRR and FAR under various p, which ranges from 0.01 to 0.06. As shown in Fig. 13, we set the value p to the value that corresponds to the intersect point of the two curves of FRR and FAR, i.e., $p = 0.03$, which is used in the following experiments.

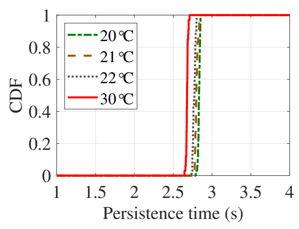


Figure 12: Impact of temperature.

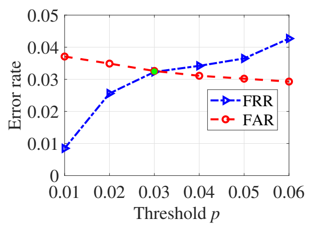


Figure 13: Threshold setting.

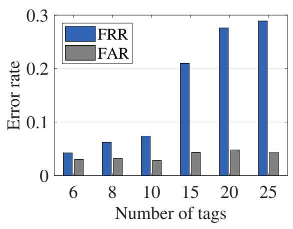


Figure 15: Multiple tags.

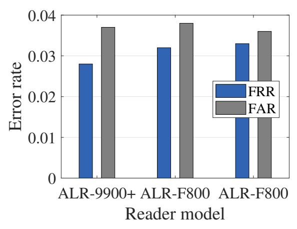


Figure 16: Device diversity.

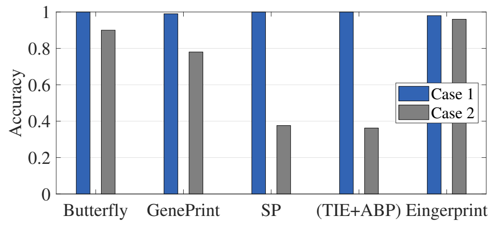


Figure 14: Performance comparison.

5.3.2 Authentication Accuracy

We now compare the authentication accuracy of Eingerprint with the state-of-the-art, including Butterfly [11], GenePrint [10], spectral feature (SP) [35], and time interval error together with the average base band power (TIE+ABP) [35]. Two cases are taken into account. In case 1, the tags are registered and authenticated at the same position. In case 2, the tags are registered and authenticated in different rooms. As shown in Fig. 14, all methods achieve a high authentication accuracy in case 1. However, the environmental changes in case 2 have a great impact on the performance of existing work. For example, the accuracy of SP drops sharply from 100% to 37.6%. By contrast, Eingerprint is resistant to these changes; the authentication accuracy in case 2 reaches 96.2%. Eingerprint is also more scalable than these approaches, which require a purpose-built device to measure the RF signals and cannot be deployed in a commercial RFID system. Notably, we just use one session to do the authentication. If more sessions are taken into account, the accuracy will be further improved, which will be shown next.

Selection of sessions. According to the Gen2 standard, three sessions with different persistence times can be used for tag authentication: session 1 (S1), session 2 (S2), and session 3 (S3). As shown in Table 3, we increasingly use these three sessions. The authentication accuracy improves as the number of sessions increases. This result is intuitive because more fingerprints reduce the probability that the system incorrectly accepts a counterfeit. Using multiple sessions, however, increases the authentication time. Hence, it is a trade-off between the accuracy and the time efficiency.

Table 3: Accuracy with different sessions

	S1	S1+S3	S1+S2+S3
Accuracy	97.3%	98.3%	99.4%

Multiple tags. We now study the performance of Eingerprint when authenticating multiple tags concurrently. We randomly choose 6, 8, 10, 15, 20, and 25 tags from 200 tags and authenticate them concurrently. As shown in Fig. 15, FRR sees a sharp rise as the number of tags increases because the large number lowers the sampling rate of each tag, which further lowers the resolution of the measured persistence time. In other words, the same tag is likely to have some persistence times apart from each other due to the low resolution, which increases the probability that a genuine tag is rejected. In contrast, the number of tags has a much lower impact on FAR because the same tag still more easily has similar persistence times than others even though the resolution is low. In addition, we can see that our method has potential in validating multiple tags in parallel. For example, when fingerprinting 10 tags, the FRR is 7.2%, the FAR is 2.3%, and the authentication accuracy is 95.2%. We assert that the degree of parallelism is related to how fast a reader can read tags. In this experiment, the read throughput of the reader is about 150 tags/s. If a faster reader is adopted, then the degree of parallelism could be higher.

Device diversity. In practice, using different devices to register and validate tags is common. To study the impact of device diversity, four readers are used, namely, three ALR-F800 readers and an ALR-9900+ reader. In the experiment, we first register 200 tags with an ALR-F800 reader and then validate the tags with the other three readers. As shown in Fig. 16, the authentication accuracy remains almost unchanged, regardless of which reader is used. This experimental result shows that device diversity has little impact on the performance of Eingerprint.

Tag model. We further study the performance of Eingerprint on different tag models. In the experiment, we test eight tag models from three leading RFID tag providers, which are Alien [1], NXP [3], and Impinj [2]. As shown in Table 4, Eingerprint achieves a high authentication accuracy (>94%) on all Alien and NXP tags. However, for Impinj tags, the accuracy experiences a sharp drop. By checking the persistence time, we find that the difference of persistence times of Impinj tags is much smaller than that of the other two. Hence, we recommend using Alien tags or NXP tags if tag authentication is required.

6 Related Work

Existing studies on RFID authentication can be divided into two categories: cryptographic-based approach [5, 8, 9, 18, 19, 32] and physical-layer identification (PLI) [6, 7, 10–12, 21, 26, 29, 31, 35–37, 40]. The former uses cryptographic technique to protect the communication between reader and tags against eavesdropping. However, existing cryptographic-based approaches suffer from two limitations. First, some cryptographic algorithms require high computation overhead, which is too heavy to be afforded by a passive tag [8]. Besides, it increases the cost of a passive tag as well as reduces the communication range between a reader and a tag. Second, some cryptographic-based methods are vulnerable to protocol-layer attacks, such as reverse engineering, side-channel, replay attack, and cloning [18, 32].

PLI is commonly referred to as RF fingerprinting, which utilizes the physical-layer information to identify digital devices. PLI has two advantages over cryptographic-based methods. First, the feature from the physical layer is unique and unpredictable, such that it can provide high security guarantees against various protocol-layer attacks. Second, no hardware or firmware upgrades on existing systems are required. Existing PLI work generally has three categories: location-based RF fingerprinting (LRF) [26, 31, 37], transient-based and preamble-based RF fingerprinting (TPF) [7, 10, 11, 29], and modulation error-based RF fingerprinting (MEF) [6, 12, 35].

LRF takes the location information as the fingerprint to authenticate a target, which works but strongly relies on the target's location. TPF fingerprints a device through the uniqueness of a certain fixed segment extracted from its transition signals and preamble signals [7, 10, 11, 29]. Since the transient-based and preamble-based features are always derived by spectral transformations, this approach is sensitive to environmental changes [29, 39]. MEF fingerprints a device through the modulation errors caused by hardware imperfection, such as SYNC correlation [6], carrier frequency offset [12], and time interval errors [35], which is channel-robust but usually requires a purpose-built device (e.g., USRP) to acquire fine-gain signal features and is thus not scalable to a commodity RFID system.

Table 4: Performance on different tag models

Company	Chip	Model	Accuracy
Alien	Higgs 3	ALN-9634	97.3%
	Higgs 4	ALN-9740	96.9%
	Higgs EC	ALN-9830	96.6%
NXP	Ucode G2iL	MiniWeb	94.4%
	Ucode G2iM	AD-380iM	94.9%
	Ucode 8	AD-238U8	94.2%
Impinj	Monza 4	H47	77.8%
	Monza R6	BLING	80.4%

7 Conclusion

In this paper, we propose a robust RFID authentication scheme by using an energy-related fingerprint. The basic idea is using the electric energy stored in a tag's circuit rather than RF signals as the fingerprint, which is resistant to the environmental changes. Directly measuring the tag's circuit to obtain the stored energy is impractical. Instead, we find an equivalent metric, namely, persistence time, that can reflect the circuit diversity indirectly. We design a Gen2-compatible select-query method to measure the persistence time. Afterwards, we use a t-test based model to validate the genuineness of a tag. We set up a prototype of the fingerprinting system, and extensive experiment results show that our system is able to achieve a high authentication accuracy of 99.4%, regardless of environmental conditions and without any hardware or firmware modifications.

Acknowledgments

We would like to thank our shepherd, Prof. Aaron Schulman, and the anonymous reviewers for their valuable feedback on an earlier draft of this paper. This work is supported by National Natural Science Foundation of China (Nos. 61702257 and 61771236), Natural Science Foundation of Jiangsu Province (BK20170648), Jiangsu Key R&D Plan (Industry Foresight and Common Key Technology, BE2017154), Fundamental Research Funds for the Central Universities, and Collaborative Innovation Center of Novel Software Technology and Industrialization. Jia Liu and Lijun Chen are the corresponding authors.

References

[1] Alien. `http://www.alientechnology.com`.

[2] Impinj. `http://www.impinj.com`.

[3] NXP. `https://www.nxp.com`.

[4] *GS1 EPCglobal. EPC radio-frequency identity protocols generation-2 UHF RFID version 2.0.1*, 2015.

[5] Karim Baghery, Behzad Abdolmaleki, Shahram Khazaei, and Mohammad Reza Aref. Breaking anonymity of some recent lightweight RFID authentication protocols. *Wireless Networks*, 25(3):1235–1252, 2019.

[6] Vladimir Brik, Suman Banerjee, Marco Gruteser, and Sangho Oh. Wireless device identification with radiometric signatures. In *Proc. of ACM MobiCom*, pages 116–127, 2008.

[7] Songlin Chen, Feiyi Xie, Yi Chen, Huanhuan Song, and Hong Wen. Identification of wireless transceiver devices using radio frequency RF fingerprinting based on

STFT analysis to enhance authentication security. In *Proc. of IEEE EMC+SIPI*, pages 1–5, 2017.

[8] Jung Sik Cho, Sang-Soo Yeo, and Sung Kwon Kim. Securing against brute-force attack: A hash-based R-FID mutual authentication protocol using a secret value. *Computer Communications*, 34(3):391–397, 2011.

[9] Lijun Gao, Maode Ma, Yantai Shu, and Yuhua Wei. An ultralightweight RFID authentication protocol with CR-C and permutation. *Journal of Network and Computer Applications*, 41:37–46, 2014.

[10] Jinsong Han, Chen Qian, Panlong Yang, Dan Ma, Zhiping Jiang, Wei Xi, and Jizhong Zhao. GenePrint: Generic and accurate physical-layer identification for UHF RFID tags. *IEEE/ACM Transactions on Networking*, 24(2):846–858, 2015.

[11] Jinsong Han, Chen Qian, Yuqin Yang, Ge Wang, Han Ding, Xin Li, and Kui Ren. Butterfly: Environment-Independent physical-layer authentication for passive RFID. In *Proc. of ACM UbiComp*, pages 1–21, 2018.

[12] Jingyu Hua, Hongyi Sun, Zhenyu Shen, Zhiyun Qian, and Sheng Zhong. Accurate and efficient wireless device fingerprinting using channel state information. In *Proc. of IEEE INFOCOM*, pages 1700–1708, 2018.

[13] Kiran Joshi, Dinesh Bharadia, Manikanta Kotaru, and Sachin Katti. WiDeo: Fine-grained device-free motion tracing using RF backscatter. In *Proc. of USENIX NS-DI*, pages 189–204, 2015.

[14] Laird. S9028PCL. `https://www.lairdtech.com/products/s9028pcl`.

[15] Jia Liu, Xingyu Chen, Xiulong Liu, Xiaocong Zhang, Xia Wang, and Lijun Chen. On improving write throughput in commodity RFID systems. In *Proc. of IEEE INFOCOM*, pages 1522–1530, 2019.

[16] Xiulong Liu, Jiannong Cao, Yanni Yang, Wenyu Qu, X-ibin Zhao, Keqiu Li, and Didi Yao. Fast RFID sensory data collection: Trade-off between computation and communication costs. *IEEE/ACM Transactions on Networking*, 27(3):1179–1191, 2019.

[17] Xuan Liu, Bin Xiao, Feng Zhu, and Shigeng Zhang. Let's work together: Fast tag identification by interference elimination for multiple RFID readers. In *Proc. of IEEE ICNP*, pages 1–10, 2016.

[18] Li Lu, Jinsong Han, Lei Hu, Yunhao Liu, and Lionel M Ni. Dynamic key-updating: Privacy-preserving authentication for RFID systems. In *Proc. of IEEE PerCom*, pages 13–22, 2007.

[19] Chen Min and Shigang Chen. ETAP: Enable lightweight anonymous RFID authentication with O(1) overhead. In *Proc. of IEEE ICNP*, pages 267–278, 2015.

[20] Senthilkumar Chinnappa Gounder Periaswamy, Dale R Thompson, and Jia Di. Fingerprinting RFID tags. *IEEE Transactions on Dependable and Secure Computing*, 8(6):938–943, 2010.

[21] Adam C Polak, Sepideh Dolatshahi, and Dennis L Goeckel. Identifying wireless users via transmitter imperfections. *IEEE Journal on Selected Areas in Communications*, 29(7):1469–1479, 2011.

[22] Saiyu Qi, Yuanqing Zheng, Mo Li, Yunhao Liu, and Jinli Qiu. Scalable data access control in RFID-enabled supply chain. In *Proc. of IEEE ICNP*, pages 71–82, 2014.

[23] Kristin Rasmussen. *Encyclopedia of measurement and statistics*, volume 1. Sage, 2007.

[24] Longfei Shangguan and Kyle Jamieson. The design and implementation of a mobile RFID tag sorting robot. In *Proc. of ACM MobiSys*, pages 31–42, 2016.

[25] Longfei Shangguan, Zheng Yang, Alex X Liu, Zimu Zhou, and Yunhao Liu. Relative localization of RFID tags using spatial-temporal phase profiling. In *Proc. of USENIX NSDI*, pages 251–263, 2015.

[26] Jitendra K Tugnait and Hyosung Kim. A channel-based hypothesis testing approach to enhance user authentication in wireless networks. In *Proc. of IEEE COMSNET-S*, pages 1–9, 2010.

[27] Chuyu Wang, Lei Xie, Keyan Zhang, Wei Wang, Yanling Bu, and Sanglu Lu. Spin-Antenna: 3D motion tracking for tag array labeled objects via spinning antenna. In *Proc. of IEEE INFOCOM*, pages 1–9, 2019.

[28] Ju Wang, Liqiong Chang, Omid Abari, and Srinivasan Keshav. Are RFID sensing systems ready for the real world? In *Proc. of ACM MobiSys*, pages 366–377, 2019.

[29] Wenhao Wang, Zhi Sun, Sixu Piao, Bocheng Zhu, and Kui Ren. Wireless physical-layer identification: Modeling and validation. *IEEE Transactions on Information Forensics and Security*, 11(9):2091–2106, 2016.

[30] Teng Wei and Xinyu Zhang. Tracking orientation of batteryless internet-of-things using RFID tags. In *Proc. of ACM MobiCom*, pages 483–484, 2016.

[31] Liang Xiao, Larry Greenstein, Narayan Mandayam, and Wade Trappe. Fingerprints in the ether: Using the

physical layer for wireless authentication. In *Proc. of IEEE ICC*, pages 4646–4651, 2007.

[32] Lei Yang, Jinsong Han, Yong Qi, and Yunhao Liu. Identification-free batch authentication for RFID tags. In *Proc. of IEEE ICNP*, pages 154–163, 2010.

[33] Lei Yang, Pai Peng, Fan Dang, Cheng Wang, Xiang Yang Li, and Yunhao Liu. Anti-counterfeiting via federated RFID tags' fingerprints and geometric relationships. In *Proc. of IEEE INFOCOM*, pages 1–9, 2015.

[34] Jihong Yu, Wei Gong, Jiangchuan Liu, and Lin Chen. Fast and reliable tag search in large-scale RFID systems: A probabilistic tree-based approach. In *Proc. of IEEE INFOCOM*, pages 1133–1141, 2018.

[35] Davide Zanetti, Boris Danev, and Srdjan Capkun. Physical-layer identification of UHF RFID tags. In *Proc. of ACM MobiCom*, pages 353–364, 2010.

[36] Davide Zanetti, Pascal Sachs, and Srdjan Capkun. On the practicality of UHF RFID fingerprinting: How real is the RFID tracking problem? In *Proc. of Springer PETS*, pages 97–116, 2011.

[37] Wondimu K Zegeye, Seifemichael B Amsalu, Yacob Astatke, and Farzad Moazzami. WiFi RSS fingerprinting indoor localization for mobile devices. In *Proc. of IEEE UEMCON*, pages 1–6, 2016.

[38] Yan Zhang, Laurence T Yang, and Jiming Chen. *RFID and Sensor Networks: Architectures, Protocols, Security, and Integrations*. CRC Press, 2009.

[39] Tianhang Zheng, Zhi Sun, and Kui Ren. FID: Function modeling-based data-independent and channel-robust physical-layer identification. In *Proc. of IEEE INFOCOM*, pages 199–207, 2019.

[40] Anding Zhu and Thomas J Brazil. Behavioral modeling of RF power amplifiers based on pruned volterra series. *IEEE Microwave and Wireless components letters*, 14(12):563–565, 2004.

LocAP: Autonomous Millimeter Accurate Mapping of WiFi Infrastructure

Roshan Ayyalasomayajula, Aditya Arun, Chenfeng Wu, Shrivatsan Rajagopalan,
Shreya Ganesaraman, Aravind Seetharaman, Ish Kumar Jain, and Dinesh Bharadia
(roshana, aarun, chw357, s1rajago, sganesar, arseetha, ikjain, dineshb)@ucsd.edu
University of California, San Diego

Abstract

Indoor localization has been studied for nearly two decades fueled by wide interest in indoor navigation, achieving the necessary decimeter-level accuracy. However, there are no real-world deployments of WiFi-based user localization algorithms, primarily because these algorithms are infrastructure dependent and therefore assume the location of the access points, their antenna geometries, and deployment orientations in the physical map. In the real world, such detailed knowledge of the location attributes of the access Point is seldom available, thereby making WiFi localization hard to deploy. In this paper, for the first time, we establish the accuracy requirements for the location attributes of access points to achieve decimeter level user localization accuracy. Surprisingly, these requirements for antenna geometries and deployment orientation are very stringent, requiring millimeter level and sub-10° of accuracy respectively, which is hard to achieve with manual effort. To ease the deployment of real-world WiFi localization, we present LocAP, which is an autonomous system to physically map the environment and accurately locate the attributes of existing wireless infrastructure in the physical space down to the required stringent accuracy of 3 mm antenna separation and 3^o deployment orientation median errors, whereas state-of-the-art algorithm reports 150 mm and 25^o respectively.

1 Introduction

Indoor navigation requires precise indoor maps and accurate user location in these maps. Google, Bing, Apple or Open Street Maps have made considerable progress towards providing precise indoor maps for notable locations like airports and shopping malls [1–4]. On the other hand, there are two decades of research on indoor localization using WiFi infrastructure that achieve decimeter accurate user locations [22, 30, 37, 39, 50, 51, 53, 58, 59, 65–69]. Despite these innovations, we still cannot use our smartphones to navigate in these indoor environments.

The key reason for this inability is the absence of the bridge

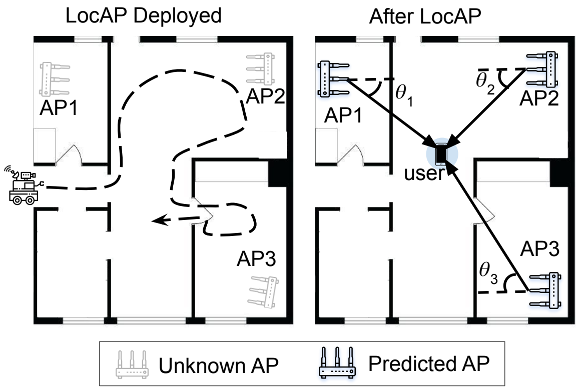


Figure 1: **Implementation of LocAP: (Left)** An unknown environment with unknown AP attributes where LocAP is deployed. **(Right)** LocAP once deployed determines the AP attributes in the physical map enabling triangulation based user localization.

providing the context of the physical map to the user locations. While there is recent work [8] that bridges this gap, it, like other state-of-the-art localization algorithms [37, 59, 66], is dependant on the accurate location attributes of the WiFi access points (APs) in the physical maps of these airports and malls. To understand what we mean by location attributes, consider the setup shown in Figure 1(right). The smartphone user is triangulated in an indoor environment by estimating the angle subtended by the user at each of the access points. This approach inherently assumes to have accurate knowledge of each access point's location and its deployment orientation (the angle at which the access point is placed in the given physical map). Further, to estimate the angle made by the user with respect to an access point, the channel state information (CSI) based WiFi localization algorithms need to know the exact antenna placements on these access points.

One can endeavor to manually locate each of these access points in the environment, but that would be labor-intensive, time-consuming and even impossible sometimes because of the following reasons. First, these access points (AP) are usually not easily visible; they may be located behind a wall or pillar. Second, even if the AP is visible, most of the access points are encased by the manufacturer, making it difficult to know the exact information of the antenna placements on the access point. Third and finally, even if we can estimate the antenna placements on the access point from the datasheet provided by the manufacturer[1], the AP's *deployment orientation* has to be carefully calibrated to the indoor maps within an error of a few degrees. Thus, we need a system that can help in accurate mapping of the existing WiFi infrastructure, which does not involve any manual labor or time.

In this paper, we present LocAP, an autonomous and accurate system to estimate access point location attributes – access point location, antenna placements, and deployment orientation. We call this process of predicting accurate access point attributes as *reverse localization*. LocAP is the first work to establish the requirements for reverse localization as follows:

Accurate Access Point Locations: As shown in Figure 2a, any error in AP location is translated to an error in the location of the user. So, any error exceeding a few tens of centimeters in access points' location is going to adversely affect the decimeter-level user localization. Thus, LocAP needs to locate the access point accurate to within tens of centimeters.

Accurate Antenna Separation: Different APs have different antenna placement configurations and the angle made by the user is measured at the access point using the spacing between antennas. So, any error in measuring antenna placements is going to cause a rotation error at the user. For example, error in antenna separation by 4 mm causes 12^o of error in the angle of user measured at the access point, which translates to up to 1 m of error for a user 5 m away from the access point. Thus, LocAP needs to predict the antenna separation accurately to within a few millimeters.

Accurate Deployment Orientation: Finally, the access points can be placed in any orientation in the environment. Any error in measurement of orientation directly translates to the predicted angle subtended by the user at the access point. Hence even 10^o of error in deployment orientation causes up to 90 cm of user location error for a user located just 5 m away from the access point. Thus, LocAP should resolve the deployment orientation of the access point accurate to less than 10^o of error.

Automation: LocAP's goal is to require no manual effort for the reverse localization, and achieve the stringent requirements discussed earlier. Furthermore, there should be zero effort to associate these positions with the existing indoor maps, ideally in an autonomous way.

LocAP achieves the aforementioned requirements and enables automated and accurate *reverse localization* of the access points. We achieve autonomy by deploying LocAP on a bot retrofitted with a multi-antenna WiFi device used in

[1]Datasheets, though publicly available do not talk about antenna placements or dimensions [6, 7, 17, 28, 47, 48, 57].

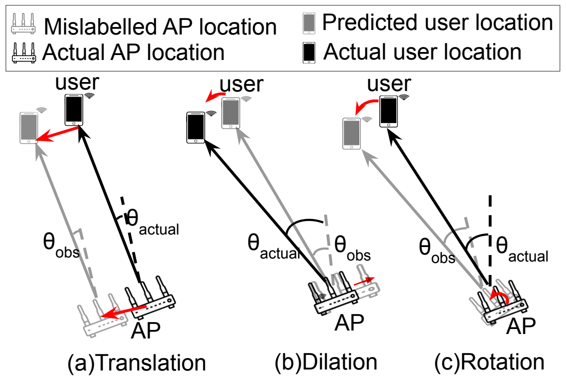


Figure 2: **LocAP's Motivation:** The user location is predicted wrong due to different errors in access point's estimated details. **(a) Translation:** Predicting the wrong location of the AP. **(b) Dilation:** Predicting wrong antenna separation on the access point results in an error in angle estimated,($\theta_{obs} \neq \theta_{exp}$) of the user. **(c) Rotation:** Predicting the wrong orientation of the AP.

[8]. When deployed in a new environment, the bot first maps the physical environment. Next, it associates with existing AP infrastructure by collecting multi-antenna channel state information, and pairing it with its predicted location in the physical map as shown in Figure 1(left). LocAP uses this information to build a database of the deployed WiFi infrastructure consisting of all the access point attributes meeting our stringent accuracy requirements. This database of accurate AP location attributes can then be used for decimeter level user localization as depicted in Fig. 1(right)

The main technical contributions of LocAP to achieve the above requirements can be summarized as follows:

cm-accurate Access Point Localization: We make an important observation that accuracy of triangulation based WiFi-localization methods improves with an increasing number of anchor points with known locations. In essence, creating an array of 100's of antennas measuring CSI at known locations achieves cm-level localization, which is not feasible in practice[2]. To overcome this, LocAP leverages the CSI data collected by the bot at 100's of predicted locations, mimicking 100's of virtual antennas with known locations. However, these *predicted* locations suffer from a varying amount of inaccuracy. Hence, LocAP designs a weighted localization algorithm, which weights each location-CSI data-point with a uniquely defined confidence metric capturing the accuracy of the predicted location.

mm-accurate Antenna Geometry Localization: We have seen earlier that both mm-error in antenna separation and error in deployment orientation lead to in-accurate Angle of Arrival (AoA) measurement at the access point, which impedes user-triangulation. Thus, LocAP tackles antenna separation and deployment orientation together by achieving millimeter-

[2]typical indoor settings are 1000-2000 sq. ft., which would imply deploying an antenna every 100 sq. ft.

level accuracy in predicting the antenna geometry. The first thought would be to use 1000's of virtual antennas to achieve cm-accurate localization [39] by locating individual antenna geometry on the AP. But, this idea can only achieve accuracy at the cm-level and will not suffice to achieve mm-level details of the antenna geometry. Our key observation is to localize the relative antenna geometry between two antennas, primarily because the relative wireless channel between the two antennas can be measured very accurately by measuring their phase information. The phase information is measured at the carrier frequency level (λ=60 mm equivalent to 360^o), hence even phase measurement accurate to 10's of degrees achieves 1-2 mm accuracy. However, this works for only relative antenna separation $d < \frac{\lambda}{2}$. LocAP designs a novel algorithm that uses relative channel information across multiple bot locations to solve for any antenna geometry, unrestricted by antenna separation, to mm-level accuracy.

Automation – Augmenting the SLAM algorithms: To avoid any manual labor and errors, LocAP is deployed on a SLAM (Simultaneous Localization and Mapping) based autonomous bot developed by us [8]. This bot provides us with a physical map and the location and heading of the bot in this physical map at all times. We pair these location-heading measurements with the CSI collected by the mounted WiFi device. However, even the best of SLAM algorithms report the location to be in-accurate up to 10-20 cm, which can have a detrimental effect on the AP location attributes. Therefore, LocAP develops a confidence metric whose core idea to look at the covariance of measurements across consecutive frames.

Further, the implementation of LocAP does not need any modification at the existing access points, as it is deployed on a custom made bot [8] that is mounted with a Quantenna client. The Quantenna client readily reports the channel-state-information (CSI) of the associated access point. We evaluate LocAP in an indoor environment of 1000 sq ft area with multiple off-the-shelf access points and 2 different antenna configurations – rectangular and linear[3] We achieved the following results satisfying the aforementioned accuracy requirements:

Relative Antenna Geometry Prediction: LocAP's relative reverse localization for the antenna separation has a median error of 3 mm (50$\times$ improvement), and a median error of 3^o(8$\times$ improvement) for deployment orientation, while state-of-the-art achieves a median error of 150 mm and 25^o respectively.

Access Point Localization: LocAP's reverse localization of the access points achieves a median localization error of 13.5 cm improving by 35% over the state-of-the-art WiFi localization algorithms [37].

Case Study-User Localization: State-of-Art user localization is deployed using the access point attributes measured manually and with LocAP. We observe user localization errors of 78 cm and 50 cm respectively, a decrease in the error of about 36%.

[3]these configurations generalize the more generic antenna deployments found on the commercial off the shelf WiFi access points.

2 Requirement and Motivation

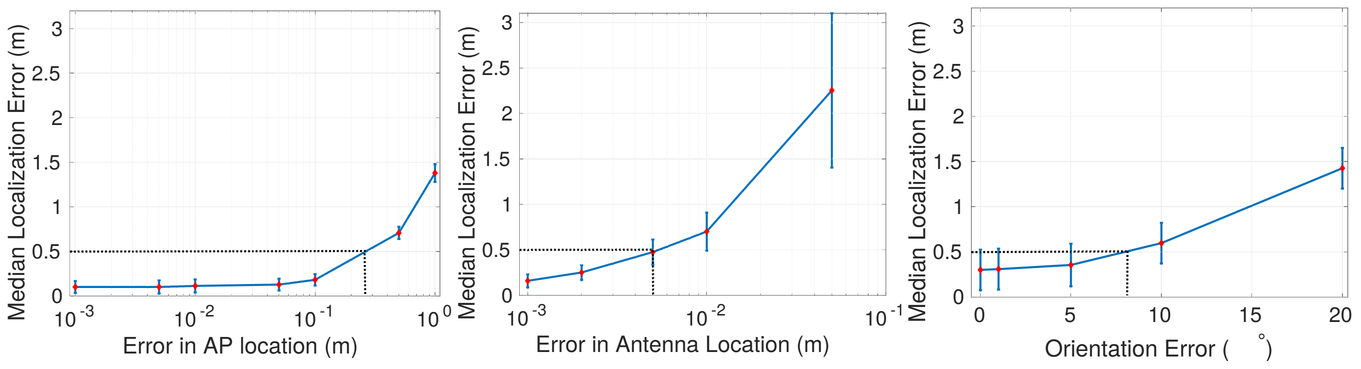
It may seem natural that user localization algorithms [37, 53, 59, 67] could be sufficient for *reverse localizing* the access point's location attributes – location, antenna geometry and deployment orientation. Surprisingly, it turns out that requirements for reverse localization of the access are stringent. To define these requirements, we conduct empirical evaluations from the standpoint on how various errors in AP attributes adversely affect the state-of-the-art decimeter level localization algorithms.

Our empirical setup contains four access points, each with 4 antennas, setup in a 25ft$\times$30ft space. The user device is placed at 100 different locations while the access points locate the user using an algorithm similar to [37]. Specifically, we aim to achieve decimeter-level localization accuracies for user WiFi localization algorithms and thus set a hardbound that no more than 50 cm median error for user localization can be tolerated.

Error in the AP's location Firstly, in the above-described setup, we incrementally increase the error in all the access points' locations. Next, we estimate the user location for each of these erroneous access point locations and calculate the user localization error. In Figure 3a, we plot the median user localization error across the access point errors reported. We can see that if the access point locations have an error of more than a few centimeters, the median localization error starts to increase. From this, we can infer that the required level of accuracy for the reverse localization of APs should be in the order of centimeters.

Error in the antenna separation Second, AoA based localization algorithms make use of the relative phase information between two antennas. Earlier, we have seen that the relative antenna position has to be estimated accurately to have exact measurements of angles. Even when the access point positions are reported correctly, we can observe that the localization error increases with just a few millimeters of errors in the reported relative antenna positions as shown in Figure 3b. This observation is intuitive because the relative antenna distances are usually of the order of a wavelength of the transmitted signal, which in the case of WiFi is 6cm. So, any error which is greater than a few millimeters is going to make a huge difference in the relative phase measured at the access point.

Error in the Deployment Orientation Finally, the antenna array can be oriented in any direction. It is also important to know the exact deployment orientation of the antenna array. Errors in this orientation will proportionately affect the angle of arrival measurements made at the access points. We observe that the greater the error in deployment orientation prediction, the higher the median localization error becomes as shown in Figure 3c. From this plot, we can see that even 7^o will degrade the median user localization accuracy to more than 50 cm.



(a) (b) (c)

Figure 3: **Robustness of localization accuracy to Access Point (AP) location errors:**(a) Shows that median localization error increases with increase in error of estimated AP location.(b) Shows how median localization error increases with increase in error of estimated antenna locations. (c) Shows how median localization error increases with increase in error of estimated antenna deployment orientation.

In summary, we should locate the access point's location with less than 30 cm of error, the antenna separation within 5 mm of error and the deployment orientation to less than 7^o of error. While these locations are typically mapped manually by humans using specialized equipment like VICON [55] or laser-based range finders [11], this process is time-consuming, labor-intensive and error-prone. So, we need a system that can accurately localize access points attributes satisfying these stringent requirements. Note that the most stringent requirements are the mm-accurate antenna separation and sub-7 degree deployment orientation. The state-of-the-art [37, 39, 53, 67] localization algorithms can locate the individual antennas to within a few 10 centimeters even by deploying hundreds of AP's in a given environment, which is insufficient to determine the antenna geometry as per required specifications established earlier in this section. Further, there are relative localization algorithms [38, 61, 64] which track the user's location across contiguous observations few millimieters apart. These ideas could potentially be used to find the relative antenna geometry. But, these tracking algorithms assume that the two relative locations are less than $\lambda/2$ apart [6, 7, 17, 28, 47, 48, 57] but the antenna separations on most access points are more than $\lambda/2$ apart, where there is an ambiguity that cannot be resolved. So, we design a system, LocAP, which fulfills these requirements and locates the access points and their antennas with the desired level of accuracy

3 Design

In this section, we present the design of LocAP. Recall that our main goal is to autonomously determine access points' location attributes within the reference coordinates of the physical map to enable easily deploy-able WiFi-localization. LocAP deploys a SLAM based autonomous bot developed in [8] to map the environment. The autonomous bot provides it's location and heading with respect to the environment's map. Simultaneously, a four antenna WiFi device retrofitted on the bot, connects with the existing WiFi infrastructure, all the while reporting the CSI information at each instance. Furthermore, to avoid changes to deployed AP infrastructure, we perform all the processing on the bot. LocAP, therefore, is provided with the location and orientation of the bot with respect to the physical map and the CSI data from the WiFi device on the bot, which connects with the existing WiFi infrastructure. We design LocAP to use these inputs to provide accurate access point attributes–location, antenna separation, and deployment orientation with respect to the physical map.

First, we discuss how to achieve the cm-level accurate location of the AP that also accounts for inaccuracies in reported bot poses. Second, we present LocAP's algorithm to estimate the antenna separation and deployment orientation of all the APs that needs to achieve the stringent requirement of mm-level accuracy. In both of these scenarios, we assume we have the CSI corresponding to the direct path and later in Section 3.3 we discuss how we tackle the presence of multipath in the environment and recover the direct path's CSI. Finally, we present the SLAM-based bot design, which does the best effort to provide the necessary measurements mentioned above. But often, these measured poses are not accurate. So, LocAP builds an algorithm which reports a confidence metric for each measured pose. This confidence metric helps us surmount the errors in the bot locations to calculate AP location attributes.

3.1 Locating the Access Point

In this subsection, we focus on identifying the position of one of the access point's antenna. This position of the antenna would then be representative of the access point's location and we refer to this as the first antenna in the subsequent text. Recall that the access point's location has to be estimated accurately to cm-level. A simple solution can be to utilize the existing WiFi localization approaches to locate one of the antennas on the access point, which would then become the access point's location. Unfortunately, state-of-

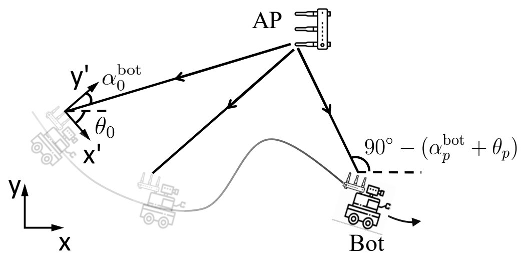


Figure 4: **First Antenna Localization:** Gives an overview of how triangulation from 10s of bot locations locates the access point accurately to within few centimeters.

the-art localization algorithms only report decimeter level location estimates. However, we make an interesting observation: these algorithms show increasing location accuracies with an increase in the number of access points deployed in an environment. In our scenario, we have a mobile bot which collects CSI data from the deployed access point at multiple anchors. This bot covers a large area setting up 100's of anchors which aids in cm-accurate first antenna localization.

Owing to this setup of LocAP, we can employ an angle of arrival estimation algorithm similar to [37] and estimate the direct path's AoA, α_p^{bot} for p^{th} bot location ($p = 1, 2, \dots, P$). We measure these AoA's with respect to the bot's local axis (X'-Y') corresponding to the first antenna's transmission for each bot location $\mathbf{u}_p = [u_p, v_p]$. To enable this AoA based first antenna triangulation, we should also know the direction of the bot's heading (θ_p) with respect to the global axis (denoted by X-Y in Figure 4), which is reported by the bot as mentioned earlier. With ($\mathbf{u}_p, \theta_p$ and α_p^{bot}) we can find the first antenna location as an intersection of P lines:

$$\text{Line}_p \equiv (y_1 - v_p) = \tan(90^\circ - (\alpha_p^{bot} + \theta_p))(x_1 - u_p) \quad (1)$$

Ironically, the AoA based triangulation accuracy is bounded by the errors in the bot's reports of its location, (u_p, v_p) and heading, θ_p^{bot}. Clearly, this creates a vicious unending loop – to predict the antenna locations we need accurate bot measurements and vice-versa to predict the bot's locations. To overcome this problem, we take advantage of SLAM algorithms [21] to get accurate ground truth estimates of the bot location and heading. Unfortunately, SLAM-based bots do not have 100% confidence in all the location estimates they report, forcing us to only cherry-pick the measurements which we believe are accurate. Based on this intuition, we design a confidence metric, $w_p \in [0, 1]$ for each bot location $\mathbf{u}_p$. Further details on the design of the confidence metric are discussed in Section 3.4. This confidence metric implies that the bot is more confident with the reported pose the closer it is to one. We thus implement a low-confidence rejection algorithm, which rejects the measurements with confidences, w_p, in the lowest 20% (Using only $\lfloor 0.8 \times P \rfloor$ lines).

We use these confidences in combination with the rest of our $\lfloor 0.8 \times P \rfloor$ line equations to define a weighted least squares problem to optimally solve for the first antenna location as follows:

$$\min_{\mathbf{x}_1} ||W(S\mathbf{x}_1 - \mathbf{t})||^2 \quad (2)$$

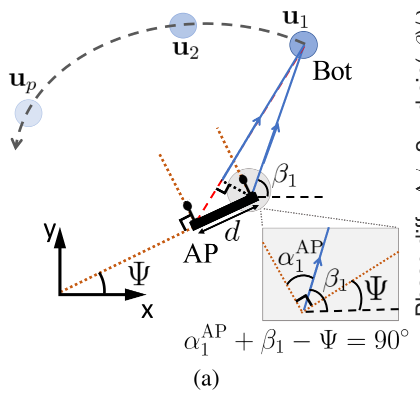
where $\mathbf{x}_1 = [x \;\; y]^T$ is the first antenna location, $W = \text{diag}(w_1, w_2, \cdots, w_{0.8P})$ is the weight matrix, $S(p,:) = [\cos(\alpha_p^{bot} + \theta_p) \;\; - \sin(\alpha_p^{bot} + \theta_p)]^T$ and $\mathbf{t}(p) = [u_p \cos(\alpha_p^{bot} + \theta_p) - v_p \sin(\alpha_p^{bot} + \theta_p)]$. Thus, we estimate of the first antenna's location $\mathbf{x}_1$ which corresponds to the access points location.

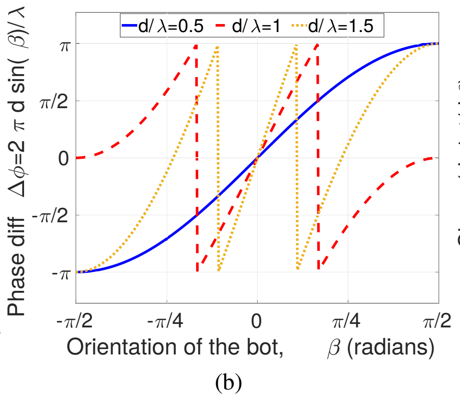
3.2 Determining Antenna Separation and Deployment orientation

As described above, we can leverage the motion of the bot to identify the accurate location of one antenna on the access point. One might wonder if it is possible to apply this algorithm iteratively to identify the location of each antenna on the access point and hence recover the relative placement of antennas. However, it is not so straightforward. In particular, the geometry prediction needs to be an order of magnitude more accurate than the location prediction. While it suffices to measure the location of the access point to cm-level, the geometry, i.e. the relative position of antennas, needs to be mm-accurate. While combining across 10s of bot locations provides antenna location accurate to cm-level, it does not extend to mm-accurate antenna geometry by combining across 100s or even 1000s of bot locations as shown in the prior art [39]. This problem occurs owing to the asynchronous clocks between the access point and the bot's WiFi device when measured at a single antenna at the access point.

To overcome this problem we make a key observation - in contrast to the phase measured at one antenna on the access point, the relative phase across two antennas is rid of synchronization errors as they share the same clock. Further, at WiFi 11ac's 5GHz carrier frequency, a wavelength of 6 cm corresponds to a phase difference of 2π radians. Empirically, we have observed that we can easily resolve phase differences up to $\pi/18$ radians (10^o), which facilitates measurement of the distance between two antennas with a resolution of 2 mm, thus enabling us to locate the antenna geometry accurately to within few millimeters. Hence, our first key insight is to measure the relative antenna separations, d_i, and deployment orientations, ψ_i, for all the N_{AP} antennas on the access point with respect to the first antenna ($i = 2, 3, \dots, N_{AP}$).

Unfortunately, although the relative phase information can resolve relative antenna separation to within 2mm, it cannot resolve for antenna separations greater than $\lambda/2$. To further understand this, consider an example scenario where the bot is moving in a circular arc about the two-antenna access point in steps of small angles as shown in Figure 5a. To avoid overcrowding of subscripts, we consider a two antenna access point and drop the access point's antenna indexing, i. Similar analysis can be performed pairwise on all the antennas on the access point with respect to the first antenna. Now, to





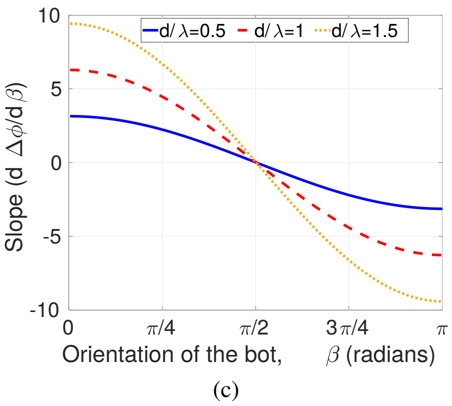


Figure 5: **Estimating AoD from phase difference:** (a) A sample case where the bot is in circular arc around the AP (b) Phase difference $\Delta\phi$ vs the orientation of the bot β assuming the deployment orientation of the AP, $\psi = 0$ (c) Slope $\frac{d\Delta\phi}{d\beta}$ vs the orientation of the bot β when compensated for the orientation of the access point.

locate the second antenna with respect to the first antenna, we analyze relative phase across these two antennas. We know that for the bot's location, $\mathbf{u}_p$, the phase difference between these two antennas corresponding to the direct path, $\Delta\phi_p$, can be estimated as:

$$\Delta\phi_p = \mod\left(\frac{2\pi d}{\lambda}\sin(90° - (\beta_p - \psi)), 2\pi\right) \quad (3)$$

where, the parameters of interest ψ and d are antenna deployment orientation (with respect to the X-axis) and antenna separation respectively. β_p is the angle subtended by the bot's location at the access point with respect to the global X-axis. From the inset in Figure 5a, we can see that the angle of departure from the AP is given by

$$\alpha_p^{AP} = 90° - (\beta_p - \psi) \quad (4)$$

and the extra distance travelled (represented by the red-dashed segment) is given by $d\sin(\alpha_p^{AP})$. This extra distance travelled induces the phase difference given in Equation 3. Thus, the phase difference across two antennas can help us estimate the antenna separation, d and deployment orientation ψ. To better understand this relation, we plot $\Delta\phi_p$ for all the bot locations along the circular arc against the angle subtended by the bot, β_p, for various antenna separations d in Figure 5b. From this plot we can see that for $d \le \lambda/2$, we have a unique mapping between the phase difference, $\Delta\phi_p$, and the bot's location, but for $d > \lambda/2$ we have ambiguous solutions that prevents us from estimating d and ψ. The ambiguity occurs because the phase difference we measured is a modulus of 2π, which means for a given $\Delta\phi_p$, the actual phase difference can be $2n_p\pi + \Delta\phi_p$, where n_p is any positive integer. This means we have three unknowns, (d, ψ, n_p) to solve for, given a single phase difference value, $\Delta\phi_p$. Furthermore, even for each additional bot location we have a new $\Delta\phi_{p+1}$ estimate, we also add an extra unknown n_{p+1} making it impossible to uniquely solve for d and ψ. LocAP's key insight is that, in contrast to the phase difference $\Delta\phi_p$, the differential phase difference with respect to the bot's angle at the AP (β_p) for optimally small increments of β_p, has a unique one-to-one mapping as shown in Figure 5c. So, the second key observation we make is that while the phase difference is not uniquely solvable for $d > \lambda/2$, the differential phase difference is uniquely solvable. Intuitively, two close bot positions will have the similar phase wrap-around's, and hence, taking the difference of the phase differences, $\Delta\phi_{p_2} - \Delta\phi_{p_1}$, can eliminate the ambiguity.

So far we have considered that the bot is moving along a circular trajectory. In fact, LocAP does not restrict the bot's motion to a circular arc and can work with arbitrary motion, as long as the CSI is measured regularly. To understand the exact implementation of LocAP's relative antenna geometry prediction, we consider a more free-flow path as shown in Figure 6. Concretely, determining the relative antenna geometry requires two parameters – the distance between antennas, d, and the deployment orientation of the antenna array, ψ, as can be seen from Figure 6. The bot moves to P distinct locations along a pre-determined trajectory about the AP and collects a series of P CSI measurements, H_p ($p = 1, 2, \cdots, P$), while simultaneously reporting the bot's locations, $\mathbf{u}_p$. The bot makes an angle β_p with respect to the global X-axis. Next, for each position of the bot, $\mathbf{u}_p$, we evaluate the differential phase difference $\frac{d\Delta\phi_p}{d\beta_p}$ between the two antennas on the access point. Differentiating Equation 3, we get

$$\frac{d\Delta\phi}{d\beta} = -\frac{2\pi d}{\lambda}\cos(90° - (\beta - \psi)) = -\frac{2\pi d}{\lambda}\sin(\beta - \psi) \quad (5)$$

But, for incremental movements of the bot, the differential phase difference in Equation 5 can be approximated as

$$\frac{d\Delta\phi_p}{d\beta_p} \approx \frac{\Delta\phi_{p+1} - \Delta\phi_p}{\beta_{p+1} - \beta_p} \quad (6)$$

The bot traces $P(> 3)$ positions as it moves, which enables us to obtain the solution from an over-determined system of equations, consequently reducing the noise level. Thus achieving highly accurate relative antenna position and orientation,

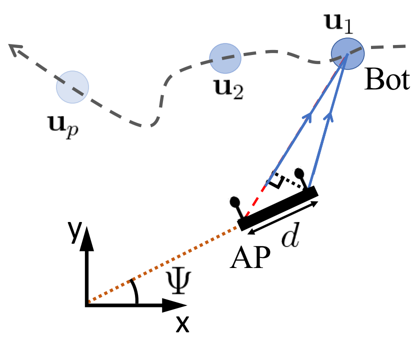


Figure 6: **Relative Geometry Prediction:** Shows the same setup as in Figure 5a with a two antenna AP making angle ψ with the positive x-axis and the bot moving about the located first antenna of the AP in an arbitrary path.

and thereby achieving millimeter-level accuracy for relative antenna localization. Now to solve for (d, ψ) uniquely as an over-determined system, it is easier to work with Cartesian co-ordinates than polar coordinates. So, we fix the location of the first antenna of the AP, the antenna on the left in Figure 6, as (x_1, y_1) and represent the second antenna (x, y) defined in the global coordinate system as:

$$(x,y) = (x_1 + d\cos(\psi), y_1 + d\sin(\psi))$$

We rewrite Equation (5) in terms of (x, y) as follows:

$$\frac{d\Delta\phi}{d\beta} = \frac{2\pi}{\lambda}[-(x-x_1)\sin(\beta_p) + (y-y_1)\cos(\beta_p)] \qquad (7)$$
$$\text{for} \quad p = 1,2,\cdots,P-1$$

Next, we represent these P set of linear equations in matrix-vector form as follows,

$$A\begin{bmatrix} x-x_1 \\ y-y_1 \end{bmatrix} = \mathbf{b} \qquad (8)$$

where A is a $(P-1)\times 2$ matrix and b is a $(P-1)$ sized column vector defined as

$$A(p,:) = \begin{bmatrix} -\sin(\beta_p) & \cos(\beta_p) \end{bmatrix} \qquad (9)$$
$$\mathbf{b}(p) = \frac{\lambda}{2\pi}\frac{\Delta\phi_{p+1} - \Delta\phi_p}{\beta_{p+1} - \beta_p}, \quad p = 1,2,\ldots,P-1 \qquad (10)$$

We further denote $\mathbf{x} = \begin{bmatrix} x & y \end{bmatrix}^T$ and $\mathbf{x}_1 = \begin{bmatrix} x_1 & y_1 \end{bmatrix}^T$. We estimate $\mathbf{x}$ to the following least squares problem:

$$\min_{\mathbf{x}} \quad ||A(\mathbf{x} - \mathbf{x}_1) - \mathbf{b}||^2 \qquad (11)$$

In this way we can uniquely solve for the cartesian coordinates of the second antenna with respect to the first antenna.

Note that the two measurements $\{\beta_p, \Delta\phi_p\}$ and $\{\beta_{p+1}, \Delta\phi_{p+1}\}$ should not be very close to avoid noise amplification. On the other hand, the measurements should not be very far apart to cause an error in the estimation of the derivative. A large separation between consecutive measurements can increase the phase difference to more than 2π, thus creating discontinuities across the series of P measurements. Our experiments suggest that around $5°$ of angular separation $(\beta_{p+1} - \beta_p)$ provides the best results for an antenna separation in $d = [0, 4\lambda]$, where $\lambda = 6$cm is the minimum wavelength in the 5GHz frequency band. We emphasize the estimated value of ψ will be in the range of $0 \le \psi \le \pi$ because the orientation of the antenna array can be defined uniquely in $0 \le \psi \le \pi$.

Generalizing Equation 11, we locate the relative location of each antenna on the access point as $\mathbf{x}_i = [x_i \quad y_i]^T$, where, $i = 2, 3, \ldots, N_{AP}$, where N_{AP} is the number of antennas on the AP. We finally find the antenna separations as $d_i = \sqrt{(x_i - x_1)^2 + (y_i - y_1)^2}$, and the deployment orientation as $\psi_i = \tan^{-1}\frac{y_i - y_1}{x_i - x_1}$, for all the antennas with respect to the first antenna, $\mathbf{x}_1$. Thus, we accurately predict the location, antenna separation and deployment orientation of the access point.

3.3 Multipath

So far, in both Section 3.1 and Section 3.2, we have assumed only one single path from the AP to the bot to solve for the access point attributes. However, the environment creates multipath which would cause the previous algorithms to fail by distorting the phase measurements. We leverage multipath rejection algorithm from [37] to estimate the direction of direct path for AP localization (Section 3.1) and build a novel algorithm to recover direct path phases as required in Section 3.2.

Recall from Section 3.1 that locating the first-antenna on the AP requires direct path AoA information at the bot. However, the received signal at the bot is usually a mix of signals arriving from different directions. We leverage multiple antennas on the bot along with the channel information across multiple subcarriers of the WiFi signal to identify the direct path and isolate it from other paths similar to prior art [37]. As first step, we collect $N_{bot} \times N_{sub}$ CSI-matrix (across N_{bot} bot client's antennas and N_{sub} subcarriers) as shown in Figure 7(a). We then apply 2D-FFT transform to estimate the AoA and Time-of-Flight (ToF) for each arriving path to the bot (Figure 7(b)). Finally, we estimate the direct path AoA by observing the signal, which has the least ToF. Intuitively, the direct path signal travels the shortest distance and thus has the lowest ToF. Thus, we can use these direct path AoA estimates to run our AP localization algorithm, as discussed in Section 3.1.

Note, however, that the direct path AoA information is not enough for estimating AP's antenna geometry (Section 3.2). In this case, our algorithm requires relative phase information across multiple AP antennas corresponding to the direct path signal. Our first insight is to estimate the direct path channel individually for each AP antenna and use them to re-

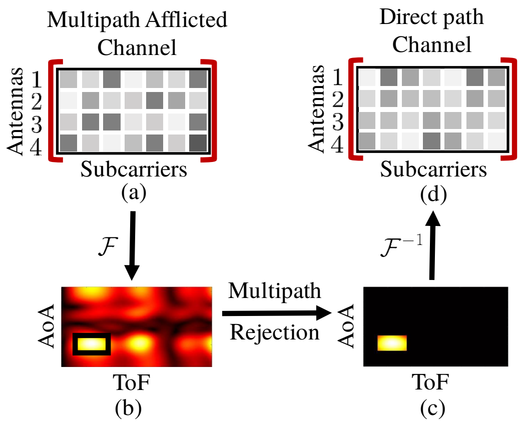


Figure 7: **Multipath rejection:** (a) Shows the measured $N_{bot} \times N_{sub}$ complex channel matrix. (b) We perform 2D FFT based transform [9] to estimate the 2D AoA-ToF profile within which we identify the direct path as the least ToF path. (c) We then perform a windowing around this peak to obtain direct path filtered AoA-ToF profile. (d) Finally, direct path's $N_{bot} \times N_{sub}$ complex channel is estimated by performing a 2D-IFFT on the windowed AoA-ToF profile.

cover the relative phase information. We take the $N_{bot} \times N_{sub}$ CSI-matrix for a fixed AP antenna and estimate the AoA-ToF profile using the same procedure as described in the previous paragraph and Figure 7(a),(b). From [37], we know that the direct path signal is concentrated around the first ToF peak (in the AoA-ToF domain). So, our insight is to apply appropriate window function in the AoA-ToF domain to remove the adulteration due to multipath (Figure 7(c)) and use this information to extract the channel corresponding to direct path. Finally, to extract the direct path signal from this windowed AoA-ToF profile, we perform 2D-IFFT on this windowed signal, as shown in Figure 7(d). As we established before, the same process can be repeated for each AP antenna to finally obtain accurate AP antenna geometries, as discussed in Section 3.2.

3.4 Autonomous Bot and Confidence Metrics

In the following section, let us look more closely at the confidence metric we mentioned in Section 3.1. We deploy RevBot largely to automate our data collection pipeline and further implementation details can be found in [8]. The key pieces of data we need to collect are the bot's pose information (provided by SLAM algorithms), and time-synchronized CSI estimates for each AP in the environment (provided by an onboard access point). Unfortunately, the position and heading reported by SLAM algorithms are not completely error-free, and the measurements can be adversely affected by the movement of the bot and the surroundings resulting in errors from 20-25 cm. These particularly worse, low-confidence measurements, need to be discarded to obtain accurate AP geometry predictions. But, most SLAM algorithms do not expose the accurate confidences of a particular reported pose. Fortunately, we can manufacture a pseudo-confidence metric by comparing the match of a current measurement with its surroundings. We make these comparisons using 3D pointclouds generated using an RGB-D camera. Pointclouds are to a 3D space what pixels are to a 2D image – each point carries an (x,y,z) coordinate and color information. We make the following observation - by looking at the registration accuracy of the point-clouds generated by consecutive pose measurements, we can estimate the quality of the relative transformation in question.

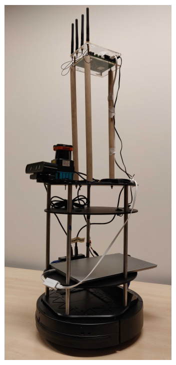
More concretely, let us consider two consecutive measurement frames $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$. We determine the relative transformation T_i between the two frames by looking at their pose estimates. Hence, T_i takes us from $\mathcal{F}_i$ to $\mathcal{F}_{i+1}$. Furthermore, from the RGB-D images captured at these frames, we can generate point-clouds. By applying T_i to the point-cloud from $\mathcal{F}_i$, we get an estimate of $\mathcal{F}_{i+1}$ and we can stitch these two point-clouds together. If T_i is accurate, then we will get a perfect overlap of these pointclouds over all the points visible in both the frames. Based on this intuition, we use the covariance matrix $\mathcal{V}_i$ as implemented by [16]. Now, this covariance matrix accommodates all six degrees of freedom as found in a 3D environment, three belonging to each direction of translation and three for each axis of rotation, hence $\mathcal{V}_i \in \mathbb{R}^{6\times 6}$. The first two diagonal elements give us the variance in the x and y position and $\mathcal{V}_i[1,2]$ gives us the co-variance between x and y. The variance in $(x+y)$ tells us how much wiggle room there is for the pose in question. Hence, the larger the wiggle room, the less confident we are in our poses. Furthermore, we observe that these variances vary in orders of magnitude, and to linearize our confidence metric, we take the log of the variance. We calculate the pseudo-confidence metric for $\mathcal{F}_i$ as

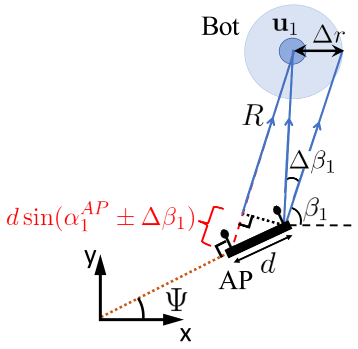
$$\begin{aligned} C_i &= \log(\text{var}(x+y)) \quad (12)\\ &= \log(\text{var}(x)+\text{var}(y)-2\text{cov}(x,y)) \quad (13)\end{aligned}$$

Finally, we normalize C_i, $\forall i = 1,2,\cdots,P$, between 0 and 1 to determine w_i, which are confidences we use in Equation 2 used to filter out the low confidence bot locations.

4 Micro-benchmarks

Before evaluating LocAP's performance, we must understand how the error in the ground truth locations reported by the autonomous bot is affecting the algorithm. We have utilized the robot implementation described in [8], while replacing the single antenna client Quantenna platform with a 4 antenna linear array Quantenna station as shown in Figure 8a. For that, we first estimate the bot's location error and analyze its effects on the accurate prediction of the location of the access point and the relative antenna geometry on the access point.


(a)



(b)

Figure 8: **Accuracy of the bot's ground truth movement:** (a) The bot we used for our experiments, a Turtlebot-2 equipped with a 4 antenna Quantenna board, LIDAR, RGB-D camera. (b) Depiction on how bot's error can effect the relative antenna localization algorithm.

4.1 Error in Bot's ground truth Location

Since, we are using the same bot setup described in [8] we use the median localization error reported for the bot in their experiments. We can observe that the median error Δr is around 6cm in this case. Further, we study the orientation errors within the same setup. We find that the median error $\Delta\beta$ in orientation is 3°.

Next, we quantify the effect of this error on the accuracy of locating the access point and determining the relative antenna geometry.

4.2 Effects of Bot's Error

First, we estimate the location of the access point. For this step, we use both the bot's location and orientation. Hence, we must look at the errors in both these measurements. We observe that an error of Δr in bot's location error directly corresponds to an error of Δr in the access point's location prediction, which is 6cm in our scenario. Next, assuming an orientation error of $\Delta\beta$, we observe that the error will be $R\Delta\beta$ in the access point's location, where R is the estimate of the distance to the access point. Hence, the upper-bound on the total error propagated will be $\Delta r + R\Delta\beta$, which for an average indoor distance of $R = 5$m would be 32cm.

Second, for the relative antenna location estimation, from Figure 8b we can see that the error in bot's location, Δr, translates to error in the angle estimated at the access point, $\beta_i + \Delta\beta_i$, where approximately $\Delta\beta_i = \frac{\Delta r}{R}$. Hence, we redefine A from Equation 9 as $A' = A\begin{bmatrix} 1 & \frac{\Delta r}{R} \\ -\frac{\Delta r}{R} & 1 \end{bmatrix}$, while b remains unchanged. Thus we can re-write Equation 11, assuming $\mathbf{x_1} = 0$, as

$$\min_{\mathbf{x}'} ||A'\mathbf{x}' - \mathbf{b}||^2 \tag{14}$$

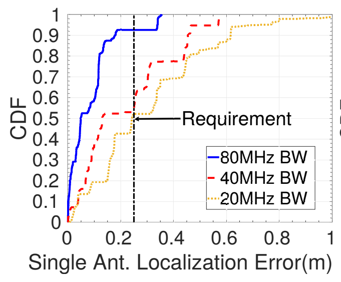
where $\mathbf{x}' = \mathbf{x} + \Delta\mathbf{x}$, and $\Delta\mathbf{x} = \begin{bmatrix}\Delta x & \Delta y\end{bmatrix}^T$. Solving for $\Delta\mathbf{x}$ from the Equations 11 and 14, and simplifying by neglecting higher order error polynomial terms we can see that $\Delta x = \frac{\Delta r}{R}y, \quad \Delta y = \frac{\Delta r}{R}x$. We know that $\mathbf{x} = [x \quad y]^T$ is of the order of few centimeters, while Δr is of the order of few centimeters and R of the order of few meters, which reduces the whole expression for Δx and Δy to be of order of $\frac{1}{10}^{th}$ millimeter, which is well within limits of the tolerance for relative antenna localization. Thus we observe that the relative antenna geometry on the access points can be estimated accurately to within few millimeters using LocAP and its implementation on our autonomous system.

5 Evaluation

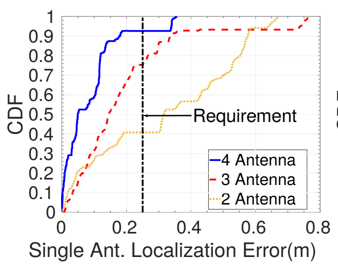
Now that we have seen all the components of LocAP, we evaluate LocAP's performance in a real world deployment to see if it has conformed to the stringent requirements we established in Section 2. For this we have deployed our autonomous bot in two different indoor environments, as shown in [8], that span 1000 sq. ft. in area, and have 8 different access points deployed at different locations, heights and orientation. Across these 8 different access points, we have covered two standard antenna geometries, linear and square antenna arrays, and covered 5 different antenna separations,$\{\lambda/2, \lambda, 3\lambda/2, 2\lambda, 5\lambda/2\}$, where $\lambda = 6$ cm is the minimum wavelength in the 155 channel of the 5GHz frequency band. Throughout this experiment, we collect CSI from multiple access points across space and time which is used to implement LocAP. The ground truth for all the evaluations are measured accurately with a commodity laser range finder [11], that is accurate up to 1mm, after carefully marking the axes on the ground and labeling the 1000 sq ft space of experimentation. This entire process of labeling the experimental space of 1000 sq ft takes a minimum of one hour spent by a group of at least three people. While there is two decades of CSI based WiFi localization, LocAP is the first work to tackle the problem of *reverse localization* of the WiFi access points and thus is compared with a state-of-the-art AoA based user localization algorithm [37], SpotFi, which combines data across multiple anchor locations.

With the given setup the overview of LocAP's results are as follows: LocAP achieves 5 cm of median localization error for the first antenna localization utilizing the weighted least squares formulation while a simple least-squares problem achieves just 8 cm of median localization error. Further, the relative geometry prediction algorithm of LocAP locates the access points in this setup accurately with a median antenna separation error of 3 mm and a median orientation error of 3^o, whereas the state-of-the-art localization algorithms achieve a 150 mm median error for antenna separation and 25° median deployment orientation error as shown in Fgure 9.

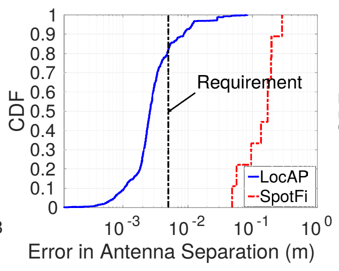
A final case study of user localization with the updated LocAP's AP attributes showed a reduction of 28 cm in me-



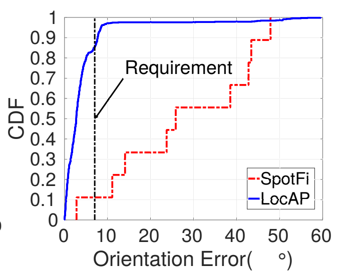
(a)



(b)



(c)



(d)

Figure 9: **Single Antenna Localization accuracy:** Shows the localization error of locating a single antenna on each AP **(a)** for various bandwidths and **(b)** for various number of antennas on the client on the autonomous bot. **(c) Antenna Separation:** CDF plot of error in measuring antenna separation across 8 different access point deployments. **(d) Deployment Orientation:** CDF plot of error in measuring deployment orientation across 8 different Access point deployments. The black vertical lines in the plots represent the requirements established in Section 2

dian user localization compared to the manual AP attribute mapping.

5.1 AP Location accuracy

To evaluate the access point localization accuracy, we deploy it in 8 different test scenarios across various heights of access points, different locations, environments and distances from the bot. To get a statistically accurate estimate of these locations, we have collected the CSI corresponding to each of these manually determined locations at 20 different time instants. With this data, we have estimated the location of each individual antenna on these access points using a least-squares triangulation algorithms employing [37]. As shown in Figure 9a, we find that the median error is 5 cm, well below the established threshold. Unfortunately, manually measuring locations takes hours of manual time and thus defeats the purpose of LocAP.

Hence, we deploy LocAP on our autonomous platform [8] that collects the same amount of data within 5 minutes. We use the SpotFi algorithm [37] as a comparative baseline model for the bot data. SpotFi assumes accurate ground truth locations of the anchors unlike LocAP's implementation that smartly rejects anchor locations that are unreliable. We observed that while the baseline model provides a median AP localization error of 20.5 cm, our weighted least squares with smart-rejection achieves 13.5 cm showing an improvement of 36% in AP localization.

Further, the bandwidth assumed for these initial results is 80MHz, while the commodity WiFi access points hardly operate at these bandwidths. These WiFi access points usually use either 20MHz or 40MHz bandwidths. To mimic this, we also collect CSI data with the same setup for both 40MHz and 20MHz bandwidths. These CSI estimates have then been utilized to test our algorithm at different WiFi bandwidths. The CDF plot for variation of localization accuracy across different bandwidths can be seen in Figure 9a. It is seen that at higher bandwidths, the localization accuracy is marginally better, while LocAP still attains centimeter-level accuracy for localizing the access point.

The design of LocAP relies on the angles estimated from the CSI data received. While the above-reported results are for a 4-antenna station, a commodity off-the-shelf WiFi device does not always have 4 antennas. Hence, we performed another experiment to observe the effect of change in the number of antennas on LocAP. This was done by changing the number of antennas present on the station mounted on the mobile robot. The CDF plot for the localization error with the increasing number of antennas can be seen in Figure 9a. The localization accuracy increases with the increasing number of antennas on the client mounted on the mobile robot. This is evidenced by the lower median error observed with 3 antennas present on the mobile robot as seen in Figure 9b. We further observe that a 2 antenna WiFi device significantly hurts the performance of LocAP. This performance degradation is because for a 2 antenna system, the multipath need to be at least 90^o apart for the two different paths to be resolved.

5.2 Relative Antenna Geometry Accuracy

After the location of the first antenna of the AP is obtained, LocAP finds the positions of the other antennas of the AP relative to the first antenna. This is achieved by traversing around the reverse localized antenna of the AP, as described in Section 3.2. To test this algorithm, we deploy APs with a linear antenna array and a square antenna array AP in the two aforementioned environments. Similar to AP location estimation, we have collected data for each antenna setup at 40 different time instances to obtain statistically accurate results. The relative antenna locations on these APs were measured using LocAP and then compared with the ground truth to get the relative antenna localization errors and the deployment orientations. We further compare these results with that derived by state-of-the-art localization algorithm, SpotFi [37].

Relative Antenna Separation: We first measure the relative antenna separation of all the antennas on the access point with respect to the first antenna and the CDF plot for the

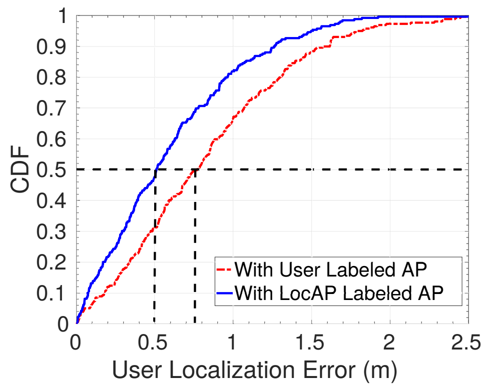


Figure 10: **User Localization accuracy:** Shows the CDF of localization accuracy after localizing the access points with LocAP and compared with those results of the manually labeled APs.

errors in relative antenna localization is shown in Figure 9c. We can see that the median error is about 3 mm for the relative antenna localization of LocAP while the state-of-the-art WiFi localization algorithm combined over multiple bot locations and time instances achieves 20 cm of median antenna separation error. Thus we show that LocAP achieves millimeter-level accuracy and meets the 5 mm error threshold set in Section 2 for predicting the antenna separation of the access point.

Deployment Orientation: We also measure the deployment orientation of all the antennas on the access point with respect to the first antenna and the CDF plot for the errors in the deployment orientation is shown in Figure 9d. We can see that while the state-of-the-art localization algorithm has a median error of 25°, LocAP's deployment orientation prediction algorithm achieves a median orientation error of just 3°, meeting the 7° limit set in Section 2.

5.3 Case Study: User Localization

So far we have seen the performance of LocAP in accurately predicting the access point attributes. We implement LocAP, to enable CSI based indoor user localization. Further LocAP is automated by deploying on a bot to remove any manual labor and time or human errors. As discussed in Section 1, human based measurements lead to high degree of errors, especially in the antenna separation measurements that are needed to be accurate to less than 5mm of errors, especially when the antennas are housed in a casing whose datasheets provided by the chip designers do not contain information regarding the antenna placements on board [28, 47, 48]. Further, the antenna placement is determined mostly by the manufacturer, and the vast cardinality of the available vendors and their models make it impossible to estimate the antenna geometry from their datasheets, which also mostly do not discuss about the antenna placements on board [6, 7, 17, 57]. Additionally, deployment orientation has to be measured accurate to less than 7° of error, which becomes extremely impossible for manual measurements. While we have shown 3mm (<5mm) error in predicting antenna separation and 3°(<7°) error in orientation deployment predictions for LocAP. To verify the effect of both LocAP mapped AP attributes and manually mapped AP attributes on the state-of-the-art indoor WiFi localization algorithms [37, 53, 67], we have asked a group of 25 people to measure the first antenna locations, relative antenna separations and the deployment orientation of the access points deployed in a realistic scenario. Users have been provided with a laser range finder [11] and compass based apps used in smartphones.

From these user measurements, we have observed that manual mapping can make their best efforts to map the AP locations accurate with 21 cm median error , the antenna separation accurate to within 4 mm of median error, and a median absolute error of about 13^o in measuring the deployment orientation of the access point.

We then deploy LocAP in a 1000 sq ft environment and locate the 4 access points' attributes. A moving user that covers 300 different marked locations in this environment is then localized using both the manually mapped and LocAP's mapped AP attributes and the corresponding CDF is shown in Figure 10c. From this plot, we can see that while human mapped AP attributes have a median localization error of 78 cm, LocAP's AP locations achieve 50 cm median error. Thus we can see that LocAP solves for the fundamental dependency of CSI based user localization algorithms by accurately predicting the AP attributes within the physical map.

6 Related Work

There has been significant work in the field of localization and LocAP's implementation work on reverse localizing the access points is closely related to the work in the following three fields:

Indoor Localization: Wide-scale deployment of WiFi based infrastructure and WiFi chips on hand-held devices makes indoor localization promising for various indoor navigation applications. There has been extensive research in WiFi based indoor localization algorithms over the past two decades [10, 15, 22, 23, 30, 37, 39, 41, 43, 45, 51, 53, 58–60, 65–69, 72, 74]. While most of the initial work was based on the Received Signal Strength Information [10, 15, 45, 60, 74] these algorithms do not achieve meter-level localization, or require extensive fingerprinting to achieve desired decimeter-level localization. Thus, most of the later work has been focused on CSI based localization algorithms [22, 30, 37, 39, 51, 53, 58, 59, 65–69]. LocAP which leverages the idea of Angle of Arrival based localization. Some such algorithms which have been developed in the past few years [37, 67] achieve decimeter-level localization and extend it to achieve centimeter-level localization accuracy. However, these WiFi based localization algorithms assume the knowledge of the location of the AP to measure the user's location with respect to the AP location. In contrast to the above work, LocAP builds a relative localization technique which provides millimeter-level accuracy for the antenna geometry on the AP. Furthermore, we also demonstrate that LocAP can solve for the antenna separation values larger than a single wavelength (λ).

Source Localization: Solving the problem of accurate knowledge of the WiFi AP locations have been attempted for RSSI based [26] and CSI based [54] systems. But these algorithms do not achieve centimeter-level localization for APs, but solve for the general regional mapping of these access points. These works are limited by the available bandwidth and thus there has also been significant work on ultra-wideband (UWB) based localization [5, 13, 14, 18, 34, 46, 49] and anchor localization algorithms [12, 19, 20, 34–36]. But these UWB systems require new infrastructure deployment. Similarly, there has been significant work towards a beacon based localization system [9, 29, 32, 33, 42, 52, 62, 63, 70, 71, 73] which have been shown to achieve decimeter-level localization but also need additional deployment of infrastructure. LocAP solves the problems of exact WiFi access point localization and exact antenna placements on WiFi Access Points.

Relative Localization: LocAP solves for millimeter-level accurate antenna placements on any given WiFi access point by borrowing and extending the principles from wireless tracking. Wireless tracking or relative localization is a well-solved problem unlike localization, with reported accuracies up to few centimeters and few millimeters [38, 61, 64]. Though all of these algorithms would need the separation between two consecutive locations to be tracked to be less than $\lambda/2$ distance apart, LocAP solves for relative localization of two antennas that are at any arbitrary distance from each other, including for distances greater than $\lambda/2$ apart. Thus LocAP can enable high mobility tracking for indoor WiFi devices.

SLAM Automation: There has been exhaustive research conducted in graph based SLAM algorithms [24]. In LocAP we employ a SLAM based autonomous bot to report ground truth and also design a metric to understand the confidence of the bot for a given ground truth. Confidences for reported measurements can be extracted from the marginal co-variances of the nodes used to describe these variables and are used to perform data association [27, 31, 44, 56]. Though these numerical methods are valid, most of them are not implemented on standard SLAM platforms, to the best of our knowledge. Furthermore, commonly used frameworks [25, 40] do not readily expose these marginal co-variances. We extend the methods described in [16] as a proxy for these internal co-variance metrics.

7 Conclusion and Future Work

We presented, LocAP, an automated reverse localization system of the existing WiFi APs that was successful in achieving the requirements for accurate localization of AP position, antenna separation and deployment orientation. After the mobile robot is allowed to traverse the unknown environment, we have a map of the indoor environment and the reverse localized positions of all the APs in this environment. If we consider the map to be part of a coordinate system, we can provide each access point with its coordinate in the environment, such that the AP becomes self-aware about its location. When a new user enters this environment, and associates with one of these APs, they can locate the user in turn almost instantaneously relative to their position.

Using the mapping and reverse localization information, we can provide accurate indoor localization and navigation for large indoor environments. These accurate AP location attributes aids many of the networking issues like user location based smart hand-off, network load balancing utilizing both AP locations and client locations and other networking services based on AP and client locations. Further, with the emergence of 5G and 11ad/ax wireless protocols, where directional beams become more and more important, these angle of arrival estimates that are provided by LocAP, can be further used to perform smart-beamforming at both the client and the AP side.

In LocAP we have analyzed the 2D scenario when the access point is in the same plane as the user to be located. In a real world deployment the access point is placed at least a meter above the user height thus subtending a non-zero polar angle at the access point. This does not affect LocAP's algorithm on relative geometry prediction as the cartesian co-ordinates defined absorb the polar angular term. Thus unchanging the formulation of the relative antenna geometry prediction algorithm enabling LocAP to perform accurately under 3D deployments.

Acknowledgements– We thank anonymous reviewers and our shepherd, Ellen Zegura, for their insightful comments and feedback. We thank Deepak Vasisht and the members of WCSNG for their comments and feedback throughout the process of the paper.

References

[1] Apple Maps. https://www.apple.com/ios/maps/.

[2] Bing Maps. www.bing.com/maps.

[3] Google Maps. www.maps.google.com.

[4] Open Street Map. www.openstreetmap.org.

[5] N. A. Alsindi, B. Alavi, and K. Pahlavan. Measurement and modeling of ultrawideband toa-based ranging in indoor multipath environments. *IEEE Transactions on Vehicular Technology*, 58(3):1046–1058, 2009.

[6] Apple. Airport Support. https://support.apple.com/airport.

[7] Aruba Networks. DS-AP303Series. https://www.arubanetworks.com/assets/ds/DS_AP303Series.pdf.

[8] R. Ayyalasomayajula, A. Arun, C. Wu, S. Sanatan, S. Abhishek, D. Vasisht, and D. Bharadia. Deep learning based wireless localization for indoor navigation. In *The 26th Annual International Conference on Mobile Computing and Networking (MobiCom '20)*. ACM, 2019.

[9] R. Ayyalasomayajula, D. Vasisht, and D. Bharadia. Bloc: Csi-based accurate localization for ble tags. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 126–138. ACM, 2018.

[10] V. Bahl and V. Padmanabhan. RADAR: An In-Building RF-based User Location and Tracking System. INFOCOM, 2000.

[11] BOSCH. Laser Measure DLE40 professional. http://www.bosch-professional.com/ma/en/laser-measure-dle-40-131500-0601016300.html.

[12] M. Cao, B. D. Anderson, and A. S. Morse. Sensor network localization with imprecise distances. *Systems & control letters*, 55(11):887–893, 2006.

[13] Y.-T. Chan, W.-Y. Tsui, H.-C. So, and P.-c. Ching. Time-of-arrival based localization under nlos conditions. *IEEE Transactions on Vehicular Technology*, 55(1):17–24, 2006.

[14] H. Chen, G. Wang, Z. Wang, H.-C. So, and H. V. Poor. Non-line-of-sight node localization based on semi-definite programming in wireless sensor networks. *IEEE Transactions on Wireless Communications*, 11(1):108–116, 2012.

[15] K. Chintalapudi, A. Padmanabha Iyer, and V. N. Padmanabhan. Indoor Localization Without the Pain. MobiCom, 2010.

[16] S. Choi, Q.-Y. Zhou, and V. Koltun. Robust reconstruction of indoor scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5556–5565, 2015.

[17] Cisco. catalyst-9120ax. https://www.cisco.com/c/en/us/products/collateral/wireless/catalyst-9120ax-series-access-points/datasheet-c78-742115.html#Aestheticallyredesignedfornextgenerationenterprise.

[18] L. Cong and W. Zhuang. Nonline-of-sight error mitigation in mobile location. *IEEE Transactions on Wireless Communications*, 4(2):560–573, 2005.

[19] C. Di Franco, A. Prorok, N. Atanasov, B. Kempke, P. Dutta, V. Kumar, and G. J. Pappas. Calibration-free network localization using non-line-of-sight ultra-wideband measurements. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 235–246. ACM, 2017.

[20] Y. Diao, Z. Lin, and M. Fu. A barycentric coordinate based distributed localization algorithm for sensor networks. *IEEE Transactions on Signal Processing*, 62(18):4760–4771, 2014.

[21] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[22] J. Gjengset, J. Xiong, G. McPhillips, and K. Jamieson. Phaser: Enabling Phased Array Signal Processing on Commodity Wi-Fi Access Points. *MobiCom*, 2014.

[23] A. Goswami, L. E. Ortiz, and S. R. Das. Wigem: A learning-based approach for indoor localization. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 3. ACM, 2011.

[24] G. Grisetti, R. Kummerle, C. Stachniss, and W. Burgard. A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, 2010.

[25] G. Grisetti, C. Stachniss, W. Burgard, et al. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE transactions on Robotics*, 23(1):34, 2007.

[26] D. Han, D. G. Andersen, M. Kaminsky, K. Papagiannaki, and S. Seshan. Access point localization using local signal strength gradient. In *International Conference on Passive and active network measurement*, pages 99–108. Springer, 2009.

[27] V. Ila, L. Polok, M. Solony, and P. Svoboda. Highly efficient compact pose slam with slam++. *arXiv preprint arXiv:1608.03037*, 2016.

[28] Intel. dual-band-wireless-ac-9260.

[29] V. Iyer, V. Talla, B. Kellogg, S. Gollakota, and J. Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *SIGCOMM*, 2016.

[30] K. Joshi, S. Hong, and S. Katti. PinPoint: Localizing Interfering Radios. NSDI, 2013.

[31] M. Kaess and F. Dellaert. Covariance recovery from a square root information matrix for data association. *Robotics and autonomous systems*, 57(12):1198–1210, 2009.

[32] B. Kellogg, A. Parks, S. Gollakota, J. R. Smith, and D. Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *ACM SIGCOMM Computer Communication Review*, 2014.

[33] B. Kellogg, V. Talla, S. Gollakota, and J. R. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *NSDI*, 2016.

[34] B. Kempke, P. Pannuto, and P. Dutta. Harmonium: Asymmetric, bandstitched uwb for fast, accurate, and robust indoor localization. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–12. IEEE, 2016.

[35] U. A. Khan, S. Kar, and J. M. Moura. Distributed sensor localization in random environments using minimal number of anchor nodes. *IEEE Transactions on Signal Processing*, 57(5):2000–2016, 2009.

[36] U. A. Khan, S. Kar, and J. M. Moura. Diland: An algorithm for distributed sensor localization with noisy distance measurements. *IEEE Transactions on Signal Processing*, 58(3):1940–1947, 2010.

[37] M. Kotaru, K. Joshi, D. Bharadia, and S. Katti. SpotFi: Decimeter Level Localization Using Wi-Fi. SIGCOMM, 2015.

[38] M. Kotaru and S. Katti. Position tracking for virtual reality using commodity wifi. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 68–78, 2017.

[39] S. Kumar, S. Gil, D. Katabi, and D. Rus. Accurate Indoor Localization with Zero Start-up Cost. MobiCom, 2014.

[40] M. Labbe and F. Michaud. Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 2019.

[41] A. M. Ladd, K. E. Bekris, A. Rudys, L. E. Kavraki, and D. S. Wallach. Robotics-based location sensing using wireless ethernet. *Wireless Networks*, 11(1-2):189–204, 2005.

[42] Y. Ma, N. Selby, and F. Adib. Drone relays for battery-free networks. In *SIGCOMM*.

[43] A. T. Mariakakis, S. Sen, J. Lee, and K.-H. Kim. Sail: Single access point-based indoor localization. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 315–328. ACM, 2014.

[44] J. Neira and J. D. Tardós. Data association in stochastic mapping using the joint compatibility test. *IEEE Transactions on robotics and automation*, 17(6):890–897, 2001.

[45] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 32–43. ACM, 2000.

[46] A. Prorok and A. Martinoli. Accurate indoor localization with ultra-wideband using spatial models and collaboration. *The International Journal of Robotics Research*, 33(4):547–568, 2014.

[47] Qualcomm. ar6004. https://www.qualcomm.com/media/documents/files/ar6004-datasheet.pdf.

[48] Quantenna. QSR10GU-AX. http://www.quantenna.com/wp-content/uploads/2018/04/QSR10GU-AX-V1.1.pdf.

[49] Z. Sahinoglu. *Ultra-wideband positioning systems*. Cambridge university press, 2008.

[50] S. Sen, J. Lee, K.-H. Kim, and P. Congdon. Avoiding multipath to revive inbuilding wifi localization. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 249–262. ACM, 2013.

[51] S. Sen, B. Radunovic, R. R. Choudhury, and T. Minka. You are facing the mona lisa: Spot localization using phy layer information. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 183–196. ACM, 2012.

[52] L. Shangguan and K. Jamieson. The design and implementation of a mobile rfid tag sorting robot. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 31–42, 2016.

[53] E. Soltanaghaei, A. Kalyanaraman, and K. Whitehouse. Multipath triangulation: Decimeter-level wifi localization and orientation with a single unaided receiver. In *MobiSyS*, 2018.

[54] A. P. Subramanian, P. Deshpande, J. Gao, and S. R. Das. Drive-by localization of roadside wifi networks. In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pages 718–725. IEEE, 2008.

[55] T-Series. VICON. www.vicon.com/products/documents/Tseries.pdf.

[56] G. D. Tipaldi, G. Grisetti, and W. Burgard. Approximate covariance estimation in graphical approaches to slam. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3460–3465. IEEE, 2007.

[57] UniFi. UAP-AC-HD-DS. https://dl.ubnt.com/datasheets/unifi/UniFi_UAP-AC-HD_DS.pdf.

[58] M. C. Vanderveen, C. B. Papadias, and A. Paulraj. Joint angle and delay estimation (jade) for multipath signals arriving at an antenna array. *IEEE Communications letters*, 1(1):12–14, 1997.

[59] D. Vasisht, S. Kumar, and D. Katabi. Decimeter-Level Localization with a Single Wi-Fi Access Point. NSDI, 2016.

[60] H. Wang, S. Sen, A. Elgohary, M. Farid, M. Youssef, and R. R. Choudhury. No need to war-drive: Unsupervised indoor localization. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 197–210, 2012.

[61] J. Wang, F. Adib, R. Knepper, D. Katabi, and D. Rus. Rf-compass: Robot object manipulation using rfids. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 3–14. ACM, 2013.

[62] J. Wang, F. Adib, R. Knepper, D. Katabi, and D. Rus. RF-compass: Robot Object Manipulation Using RFIDs. MobiCom, 2013.

[63] J. Wang, H. Jiang, J. Xiong, K. Jamieson, X. Chen, D. Fang, and B. Xie. LiFS: Low Human-effort, Device-free Localization with Fine-grained Subcarrier Information. MobiCom, 2016.

[64] J. Wang, D. Vasisht, and D. Katabi. Rf-idraw: Virtual touch screen in the air using rf signals. *ACM SIGCOMM*, 2014.

[65] Y. Xie, J. Xiong, M. Li, and K. Jamieson. xd-track: leveraging multi-dimensional information for passive wi-fi tracking. In *HotWireless*, pages 39–43. ACM, 2016.

[66] Y. Xie, J. Xiong, M. Li, and K. Jamieson. md-track: Leveraging multi-dimensionality in passive indoor wi-fi tracking. *arXiv preprint arXiv:1812.03103*, 2018.

[67] J. Xiong and K. Jamieson. ArrayTrack: A Fine-grained Indoor Location System. NSDI, 2013.

[68] J. Xiong, K. Jamieson, and K. Sundaresan. Synchronicity: Pushing the envelope of fine-grained localization with distributed mimo. In *HotWireless*, 2014.

[69] J. Xiong, K. Sundaresan, and K. Jamieson. ToneTrack: Leveraging Frequency-Agile Radios for Time-Based Indoor Wireless Localization. MobiCom , 2015.

[70] C. Xu, B. Firner, Y. Zhang, R. Howard, J. Li, and X. Lin. Improving RF-based Device-free Passive Localization in Cluttered Indoor Environments Through Probabilistic Classification Methods. IPSN, 2012.

[71] L. Yang, Y. Chen, X.-Y. Li, C. Xiao, M. Li, and Y. Liu. Tagoram: Real-time tracking of mobile rfid tags to high precision using cots devices. MobiCom, 2014.

[72] M. Youssef and A. Agrawala. The Horus WLAN Location Determination System. MobiSys, 2005.

[73] P. Zhang, D. Bharadia, K. Joshi, and S. Katti. Hitchhike: Practical backscatter using commodity wifi. In *SenSys*, 2016.

[74] X. Zhu and Y. Feng. Rssi-based algorithm for indoor localization. *Communications and Network*, 5(02):37, 2013.